

ГОСУДАРСТВЕННЫЙ СТАНДАРТ
СОЮЗА ССР

ЯЗЫК ПРОГРАММИРОВАНИЯ АДА

ГОСТ 27831-88
(ИСО 8652-87)

Издание официальное

БЗ 8–88/526

ГОСУДАРСТВЕННЫЙ КОМИТЕТ СССР ПО СТАНДАРТАМ
Москва

gost.e-ecolog.ru

УДК 681.3.06:006.354 Группа П85

ГОСУДАРСТВЕННЫЙ СТАНДАРТ СОЮЗА ССР

ЯЗЫК ПРОГРАММИРОВАНИЯ АДА

Programming language Ada

ГОСТ 27831–88

(ИСО 8652–87)

ОКСТУ 4002

Дата введения 01.07.89

Настоящий стандарт устанавливает базовое описание языка программирования АДА, применяемого для разработки программ различного назначения, в том числе работающих в реальном масштабе времени, а также средств компиляции, тестирования и отладки программ.

В процессе принятия решений при проведении испытаний и приемке систем программирования и программ на языке Ада настоящий стандарт является основанием для определения их полного соответствия языку Ада только при наличии и применении систем тестов, согласованных в установленном порядке.

1. ОБЩИЕ ПОЛОЖЕНИЯ

1.1. **Область действия стандарта**

В настоящем стандарте описаны форма представления и семантика программных модулей, написанных на языке Ада. Цель стандарта – повысить надежность и переносимость программ для разнообразных систем обработки данных.

1.1.1. Содержание стандарта

Стандарт определяет:

а) Форму представления программного модуля, написанного на языке Ада.

б) Результаты трансляции и выполнения такого программного модуля.

в) Способ получения Ада-программ из набора программных модулей.

г) Предопределенные программные модули, которые должна обеспечивать согласованная со стандартом реализация (далее – согласованная реализация).

д) Допустимые в рамках стандарта изменения языка и способы их задания.

Издание официальное ★ **Перепечатка воспрещена**

© Издательство стандартов, 1989

е) Те нарушения стандарта, которые обязана обнаруживать согласованная реализация, а также результат попытки транслировать или выполнять программный модуль, содержащий такие нарушения.

ж) Те нарушения стандарта, которые согласованная реализация не обязана обнаруживать.

Стандарт не определяет:

з) Средства преобразования программного модуля, написанного на языке Ада, в объектный код, выполняемый процессором.

и) Средства инициализации трансляции, выполнения программных модулей и управления ими.

к) Размер и скорость выполнения объектного кода, а также относительную скорость выполнения различных языковых конструкций.

л) Форму и содержание любых листингов, предусмотренных реализациями, в частности, форму и содержание сообщений об ошибках или предупреждающих сообщений.

м) Результат выполнения программного модуля, содержащего любое нарушение стандарта, которое согласованная реализация не обязана обнаруживать.

н) Предельный для конкретной согласованной реализации размер программы или программного модуля.

Там, где стандарт предписывает, что программный модуль, написанный на языке Ада, имеет точный результат, этот результат является функциональным значением данного программного модуля, и его должны обеспечивать все согласованные реализации. Там, где стандарт допускает неоднозначность результата выполнения программного модуля, под функциональным значением программного модуля в целом понимается множество возможных результатов, и согласованная реализация может обеспечить любой из возможных результатов. Примерами допустимых неоднозначностей являются:

- Значения фиксированных и плавающих числовых величин, а также результаты операций над ними;
- Порядок выполнения операторов в различных параллельных задачах при отсутствии явной синхронизации.

1.1.2. Согласованность реализации со стандартом

Согласованная реализация должна:

а) Корректно транслировать и выполнять правильные программные модули, написанные на языке Ада, если только их объем находится в допустимых для реализации пределах.

б) Не выполнять программные модули, объем которых больше допускаемого реализацией.

в) Не выполнять программные модули, содержащие ошибки, обнаружение которых предусмотрено стандартом;

г) Включать предусмотренные стандартом предопределенные программные модули.

д) Содержать только те изменения, которые допускает стандарт.

е) Описывать допустимые изменения способом, разрешенным стандартом.

1.2. **Структура стандарта**

Стандарт содержит четырнадцать глав, четыре обязательных и два справочных приложения.

Каждая глава делится на разделы, которые имеют общую структуру. В каждом разделе вводятся соответствующие понятия, даются все необходимые синтаксические правила и описывается семантика соответствующих конструкций. В конце раздела могут быть даны примеры и примечание.

Примеры предназначены для иллюстрации возможных форм описанных конструкций. Примечание предназначено для пояснения следствий из правил, описанных в данном или других разделах.

Определение стандарта языка программирования Ада предполагает следующее ограничение: материал каждого из перечисленных ниже пунктов носит информативный характер и не является частью определения стандарта языка.

- Разд. 1.3. Цели и источники разработки.
- Разд. 1.4. Обзор свойств языка.
- Примеры и примечания, приведенные в конце любого раздела.
- Каждый раздел, заголовок которого начинается со слов „Пример" или „Примеры".

1.3. **Цели и источники разработки**

Язык был разработан исходя из трех противоречивых требований:

– обеспечить надежность и сопровождение программ;

– поддерживать программирование как вид человеческой деятельности;

– обеспечить эффективность программ.

Необходимость в языках, повышающих надежность и упрощающих сопровождение программ, является установленным фактом. В языке Ада предпочтение было отдано удобочитаемости программы по сравнению с легкостью ее написания. Например, правила языка требуют, чтобы все переменные и их типы были явно описаны в программе. Так как тип переменной неизменен, компилятор может гарантировать совместимость операций над переменными со свойствами, присущими объектам этого типа. Более того, чтобы избежать обозначений, которые могут привести к ошибкам, в синтаксисе языка было отдано предпочтение конструкциям, которые ближе к естественному языку. Язык поддерживает раздельную компиляцию программных модулей способом, облегчающим разработку и сопровождение программ и обеспечивающим один и тот же уровень контроля для межмодульных и внутримодульных связей.

Человеческий фактор программирования также повлиял на разработку языка. Прежде всего была сделана попытка ограничить язык настолько, насколько это позволяла широкая область его применения. Язык охватывает всю область применения небольшим числом основных понятий путем систе-

матизации и выявления однородности, а также учитывает требование описания такой семантики, которая совпадает с интуитивным представлением о ней у пользователя.

Как и многие другие виды человеческой деятельности, разработка программ становится все более децентрализованной и разобщенной. Следовательно, одной из центральных идей при разработке языка было обеспечить возможность составлять программу из независимо разработанных компонентов. Концепции пакетов, личных типов и настраиваемых модулей прямо служат этой идее, которая повлияла на многие другие аспекты языка.

При создании любого языка необходим учет проблемы эффективности. Языки, которые требуют сверхсложных компиляторов или приводят к неэффективному использованию памяти и времени выполнения программы, дают неэффективные результаты на всех машинах и для всех программ. Каждая конструкция языка оценивалась с точки зрения современных методов реализации. Отвергались все те конструкции, которые были недостаточно ясными или требовали чрезмерных машинных ресурсов.

Ни одна из указанных выше целей разработки языка не откладывалась на будущее. Все они учитывались одновременно и с самого начала разработки языка.

При разработке любого языка трудность заключается в том, что необходимо определить не только возможности, которыми он должен обладать и которые диктует предполагаемая область применения, но и собственно разработать средства языка, обеспечивающие эти возможности. Такие возможности и требования были учтены при формировании языка.

Другое обстоятельство, существенно упростившее разработку, было следствием приобретенного ранее опыта реализации удачных проектов на базе языка Паскаль, созданного с целями, аналогичными указанным выше. Такими языками являются языки Евклид, Лис, Меса, Модула и Сью. Многие из ключевых идей и синтаксических форм этих языков имеют аналоги в языке Ада. Некоторые существующие языки, такие как Алгол 68 и Симула, а также современные проекты языков Альфард и Клу, также повлияли на разработку языка, хотя и в меньшей степени, чем языки семейства Паскаль.

1.4. **Обзор свойств языка**

Ада-программа представляет собой один или несколько программных модулей, которые могут компилироваться раздельно. Программные модули – это подпрограммы (определяющие выполняемый алгоритм), пакеты (определяющие наборы понятий), задачные модули (определяющие параллельные вычисления) или настраиваемые модули (определяющие параметризованные пакеты и подпрограммы). Каждый модуль обычно состоит из двух частей: спецификации, содержащей видимую для других модулей информацию, и тела, содержащего детали реализации, о которых другие модули не обязаны знать.

Это различие между спецификацией и телом, а также возможность компилировать модули раздельно позволяют разрабатывать, записывать и тес-

тировать программу как множество в достаточной степени независимых программных компонентов.

Ада-программа может использовать библиотеку программных модулей общего назначения.

Язык предоставляет средства, с помощью которых отдельные организации могут создавать свои собственные библиотеки. В тексте раздельно компилируемого программного модуля должны быть указаны имена библиотечных модулей, которые ему требуются.

Программные модули. Подпрограмма является основным модулем для представления алгоритма. Существуют два вида подпрограмм: процедуры и функции. Процедура – это средство вызова последовательности действий. Например, она может считывать данные, изменять значения переменных или выводить информацию. Процедура может иметь параметры для управления механизмом передачи информации между процедурой и точкой вызова.

Функция – это средство вызова действий по вычислению значения. Она подобна процедуре, но в результате выполнения еще и возвращает результат.

Пакет – это основной модуль для определения набора логически связанных понятий. Например, пакет может быть использован для определения общей группы данных и типов, набора взаимосвязанных подпрограмм или же множества описаний типов и соответствующих операций. Части пакета могут быть скрыты от пользователя, следовательно, доступ будет разрешен только к тем логическим свойствам, которые описаны в спецификации пакета.

Задачный модуль – это основной модуль для определения задачи, последовательность действий которой может выполняться параллельно с выполнением других задач. Такие задачи могут быть реализованы на многомашинной или многопроцессорной системе, либо чередованием выполнения задач на одном процессоре. Задачный модуль может определить или одну выполняемую задачу или задачный тип, позволяющий создать любое количество подобных задач.

Настраиваемый модуль – это шаблон, по которому можно получить конкретизацию в виде подпрограммы или пакета.

Описания и операторы. Тело программного модуля, как правило, содержит две части: раздел описаний, который определяет логические понятия, используемые в программном модуле, и последовательность операторов, определяющая выполнение этого программного модуля.

Раздел описаний связывает имена с описанными понятиями. Например, имя может обозначать тип, константу, переменную или исключение. Раздел описаний также может вводить имена и параметры других вложенных подпрограмм, пакеты, задачные модули и настраиваемые модули, используемые в программном модуле.

Последовательность операторов описывает последовательность действий, которые должны быть выполнены. Операторы выполняются последовательно (если только оператор возврата, перехода, выхода или возбуждения исключения не вызовет продолжения выполнения с другого места).

Оператор присваивания изменяет значение переменной. Вызов процедуры инициирует выполнение процедуры после сопоставления каждого фактического параметра, заданного в вызове, соответствующему формальному параметру.

Оператор выбора и условный оператор позволяют выполнить одну из входящих в них последовательностей операторов, определяемую значением выражения или значением условия.

Оператор цикла обеспечивает основной итерационный механизм в языке. Оператор цикла указывает, что повторение некоторой последовательности операторов выполняется по заданной итерационной схеме или до выполнения оператора выхода.

Оператор блока включает в себя последовательность операторов, которым предшествуют описания локальных понятий, используемых в этих операторах.

Некоторые операторы применимы только к задачам. Оператор задержки приостанавливает выполнение задачи на указанный интервал времени. Оператор вызова входа записывается как оператор вызова процедуры; он показывает, что выполнившая этот вызов задача готова для рандеву с другой задачей, имеющей указанный вход. Вызываемая задача готова принять вызов входа, когда ее выполнение достигает соответствующего оператора принятия, который определяет выполняемые далее действия. После завершения рандеву обе задачи как вызывающая, так и имеющая вход, продолжают свое параллельное выполнение. Одна из форм оператора отбора допускает отбор с ожиданием для одного из нескольких альтернативных рандеву. Другие формы оператора отбора допускают условные или временные вызовы входа.

Выполнение программного модуля может привести к ошибочным ситуациям, вследствие чего продолжение нормального выполнения программы невозможно. Например, когда результат арифметического вычисления превышает максимальное допустимое числовое значение или когда делается попытка доступа к компоненту массива с неправильным значением индекса. Для работы с такими ошибочными ситуациями текстуально за операторами программного модуля могут следовать обработчики исключений, определяющие предпринимаемые при возникновении ошибочных ситуаций действия. Исключения могут быть возбуждены и явно оператором возбуждения.

Типы данных. Каждый объект языка имеет тип, характеризующий множество значений и множество применимых к ним операций. Основные классы типов – это скалярные типы (включающие перечислимые и числовые типы), составные, ссылочные и личные типы.

Перечислимый тип определяет упорядоченное множество различных литералов перечисления, например, список состояний или перечень символов. Перечислимые типы BOOLEAN и CHARACTER предопределены.

Числовые типы обеспечивают средства выполнения точных или приближенных числовых вычислений. Для точных вычислений используются целые

типы, которые обозначают множество последовательных целых чисел. В приближенных вычислениях используются либо фиксированные типы (типы чисел с фиксированной точкой), представимые с абсолютной погрешностью, либо плавающие типы (типы чисел с плавающей точкой), представимые с относительной погрешностью. Числовые типы INTEGER, FLOAT и DURATION предопределены.

Составные типы допускают определения структурных объектов из сгруппированных компонентов. Составные типы в языке представлены массивами и записями. Массив – это объект с индексируемыми компонентами одного и того же типа. Запись – это объект с именованными компонентами, возможно, различных типов. Индексируемый тип STRING предопределен.

Запись может иметь специальные компоненты, называемые дискриминантами. В записях можно определить альтернативные структуры, зависящие от значений дискриминантов.

Ссылочные типы позволяют вычислением генератора создавать связанные ссылками структуры данных. Они позволяют нескольким переменным ссылочного типа указывать на один и тот же объект, а компонентам одного объекта указывать на тот же самый или другие объекты. Элементы такой связанной структуры данных и их связи с другими элементами могут быть изменены во время выполнения программы.

Личные типы могут быть определены в пакете, скрывающем внутреннюю структуру, несущественную вне пакета. Пользователю таких типов видны лишь логически существенные их свойства (включая дискриминанты).

Концепция типа уточняется концепцией подтипа, благодаря чему пользователь может ограничить набор допустимых значений данного типа. Подтипы могут быть использованы для определения поддиапазонов скалярных типов, массивов с ограниченным множеством значений индексов, а также именованных и личных типов с конкретными значениями дискриминантов.

Другие средства языка. Для определения отображения типов на архитектуру объектной машины можно использовать спецификатор представления. Например, пользователь может задать число битов для представления объектов данного типа или размещение в памяти машины компонентов записи. Другие свойства языка допускают управляемое использование особенностей, связанных с низким уровнем, непереносимостью или зависимостью от реализации, включая прямое использование машинного кода.

Ввод-вывод в языке определен средствами предопределенных библиотечных пакетов. Предоставляются средства для ввода-вывода значений типов, определенных пользователем, а также значений предопределенных типов; обеспечивается представление изображений значений в стандартной форме.

Наконец, язык предоставляет мощные средства параметризации программных модулей, называемых настраиваемыми программными модулями. Параметрами настройки могут быть типы и подпрограммы (а также объекты), и, таким образом, допустимы общие алгоритмы, применимые для всех типов данного класса.

1.5. **Метод описания и синтаксические обозначения**

Контекстно-свободный синтаксис программных модулей языка Ада вместе с контекстно-зависимыми требованиями выражаются в повествовательной форме.

Семантика программных модулей описана правилами определения результата выполнения каждой конструкции и правилами их построения. В изложении используются термины, точное определение которых дано в тексте.

Все другие понятия имеют свое естественное значение, определенное в словаре русского языка Ушакова*.

Контекстно-свободный синтаксис языка описывается с помощью простого варианта форм Бэкуса-Наура, в частности:

а) Записанные строчными буквами слова, возможно содержащие в некоторых случаях символ подчеркивания, используются для обозначения синтаксических понятий, например:

аддитивная_операция

В названиях синтаксических понятий, используемых вне контекста синтаксических правил, вместо символа подчеркивания используется пробел, например:

аддитивная операция

б) Полужирным шрифтом выделены зарезервированные слова, например:

array

в) В квадратные скобки заключены необязательные элементы. Поэтому два следующих правила эквивалентны:

оператор возврата : : = **return** [выражение] ;

оператор возврата : : = **return** ; | **return** выражение;

г) Повторяющиеся элементы заключаются в фигурные скобки. Этот элемент может встретиться нуль или более раз; повторение осуществляется слева направо в соответствии с правилом левой рекурсии. Таким образом, два следующих правила эквивалентны:

слагаемое : : = множитель {операция_умножения множитель}

слагаемое : : = множитель | слагаемое операция_умножения множитель

д) Вертикальная черта разделяет альтернативные элементы, кроме случаев, когда черта встречается непосредственно за открывающей фигурной скобкой, тогда она обозначает знак вертикальной черты:

буква_или_цифра : : = буква|цифра

сопоставление_компонентов : : =

[выбор {|выбор} = >] выражение

е) Если название какого-нибудь синтаксического понятия содержит выделенную курсивом часть, оно эквивалентно названию синтаксического по-

---

* Толковый словарь русского языка/Под ред. Д.Н. Ушакова. – М.: Государственное издательство иностранных и национальных словарей, 1938.

нятия без выделенной курсивом части. Выделенная курсивом часть предназначена для выражения некоторой семантической информации. Например, имя_*типа* и имя_*задачи* эквивалентны просто понятию имя.

Примечание. Описывающие структурные конструкции синтаксические правила представлены в форме, соответствующей рекомендованному делению на абзацы. Например, условный оператор определяется так:

```
условный оператор : : =
  if условие then
      последовательность_операторов
  {elsif условие then
      последовательность_операторов}
  [else
      последовательность_операторов]
  end if;
```

Синтаксические правила записываются в несколько строк, если соответствующие части конструкции рекомендуется располагать на разных строчках. Все отступы от начала строчки рекомендованы в правилах для сдвига соответствующих частей конструкции. Все отступы должны быть кратны базовому шагу отступа (число пробелов в базовом шаге не определяется). Переход на новую строчку рекомендуется после точки с запятой. Допускается размещение всей конструкции в одной строчке.

1.6. **Классификация ошибок**

Определение языка делит ошибки на несколько различных категорий:

а) Ошибки, которые должны быть обнаружены во время компиляции любым компилятором с языка Ада.

Эти ошибки соответствуют любому нарушению правил, данных в этом стандарте, кроме нарушений, соответствующих подпунктам *б* и *в*. В частности, к этой категории относятся нарушения правил, в которых использованы слова *должно, допустимо* (или *недопустимо*), *правильный* или *неправильный*. Любая содержащая такую ошибку Ада-программа не является правильной; с другой стороны, тот факт, что программа правильна в этом смысле, не означает, что в ней нет других ошибок.

б) Ошибки, которые должны быть обнаружены во время выполнения Ада-программы.

Соответствующим ошибочным ситуациям сопоставлены имена предопределенных исключений. Каждый компилятор с языка Ада должен генерировать код, возбуждающий соответствующее исключение, если такая ошибочная ситуация обнаружится во время выполнения программы. Если исключение обязательно будет возбуждаться при выполнении данной программы, то компиляторы могут (но не обязательно) сообщить об этом во время компиляции.

в) Ошибочное выполнение.

В языке определен ряд правил, которым должна подчиняться Ада-программа, хотя от компилятора и не требуется обнаружение нарушений этих правил ни во время компиляции, ни во время выполнения программы. Слово *ошибочный* квалифицирует выполнение конструкций, содержащих ошибки этой категории. Результат выполнения ошибочной конструкции – непредсказуем.

г) Некорректная зависимость от порядка выполнения.

Когда в стандарте указывается, что различные части данной конструкции должны быть выполнены *в порядке, который не определен в языке,* это означает, что реализация может выполнять эти части в любом порядке, но не параллельно. Более того, конструкция некорректна, если выполнение этих частей в различном порядке дает различный результат. Во время компиляции и во время выполнения программы (этот процесс называется *выполнением*) компилятор не обязан обеспечивать проверку некорректной зависимости результата от порядка. Термин *выполнение* в равной мере применим к процессам, которые называются вычислением и предвыполнением.

Если компилятор способен распознать во время компиляции, что конструкция ошибочна или содержит некорректную зависимость от порядка, то допускается, чтобы компилятор генерировал код, заменяющий конструкцию кодом, возбуждающим предопределенное исключение PROGRAM_ERROR. Компилятор также может сгенерировать код, который во время выполнения проверяет ошибочность конструкции, некорректную зависимость от порядка или и то и другое. Предопределенное исключение PROGRAM_ERROR возбуждается, если проверка покажет наличие такой ошибки.

2. ЛЕКСИКА

Текст программы состоит из текстов одной или нескольких компиляций.Текст компиляции – это последовательность лексических элементов (лексем), каждый из которых состоит из символов. В этой главе приведены правила составления лексем. Кроме того, в ней описаны прагмы, задающие определенную информацию для компилятора.

2.1. **Набор символов**

Символами текста программы должны быть только графические символы и символы управления форматом. Каждый графический символ представляется (визуально) графическим знаком. Описание определения языка в этом документе использует стандартный набор графических символов по ГОСТ 27465–87.

графический символ : : = основной_графический_символ
| строчная_буква | дополнительный_специальный_символ
основной_графический_символ : : = прописная_буква
| цифра | специальный_ символ| символ _пробела
основной_символ : : = основной_графический_символ
| символ_управления_форматом

Набор основных символов достаточен для написания любой программы. Основные графические символы подразделяются на следующие подклассы:

а) прописные буквы

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

А Б В Г Д Е Ё Ж З И Й К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я

б) цифры

0 1 2 3 4 5 6 7 8 9

в) специальные символы

" # & ' ( ) * + , – . / : ; < = > _ |

г) символ пробела

Символы управления форматом (часть символов формата) – это символы по ГОСТ 27465–87, называющиеся: горизонтальная табуляция, вертикальная табуляция, возврат каретки, перевод строчки и перевод формата.

Остальные подклассы графических символов определяются следующим образом:

д) строчные буквы

a b c d e f g h i j k l m n o p q r s t u v w x y z

а б в г д е ё ж з и й к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я

е) дополнительные специальные символы

! ¤ % ? @ [\ ] ^ ` { } ~

В разд. 2.10 определены допустимые замены для специальных символов вертикальной черты (|), номера (#) и кавычки (").

Примечание. Шрифтовые выделения графических символов (например, курсив или полужирный шрифт) не являются частью стандартного набора.

Для ссылок на специальные и дополнительные специальные символы в табл. 2.1 приведены их наименования.

Таблица 2.1

Символ	*Наименование*	*Символ*	*Наименование*
"	Кавычки	>	Больше
#	Номер	_	Подчеркивание
&	Коммерческое И	|	Вертикальная черта
'	Апостроф	!	Восклицательный знак
(	Круглая скобка левая	¤	Знак денежной единицы
)	Круглая скобка правая	%	Проценты
*	Звездочка	?	Вопросительный знак
+	Плюс	@	Коммерческое ЭТ
,	Запятая	[	Квадратная скобка левая
–	Минус	\	Обратная дробная черта
.	Точка	]	Квадратная скобка правая
/	Дробная черта	^	Сиркюмфлекс
:	Двоеточие	`	Слабое ударение
;	Точка с запятой	{	Фигурная скобка левая
<	Меньше	}	Фигурная скобка правая
=	Равно	~	Черта сверху

2.2. Лексемы, разделители и ограничители

Текст каждой компиляции – это последовательность отдельных лексем. Лексема (лексический элемент) – это ограничитель, идентификатор (который может быть зарезервированным словом), числовой литерал, символьный литерал, строковый литерал или комментарий. Результат выполнения программы зависит только от конкретной последовательности лексем, исключая возможные комментарии.

В некоторых случаях необходим явный *разделитель* между соседними лексемами (в противном случае они могут быть восприняты как одна).

Разделителем может быть символ пробела, символ управления форматом или конец строчки. Символ пробела не является разделителем в комментарии, строковом литерале или в символьном литерале. Символ управления форматом (кроме символа горизонтальной табуляции, когда он употребляется в комментариях) всегда является разделителем.

Конец строчки всегда является разделителем. Язык не определяет, что является концом строчки. Однако, если в данной реализации конец строчки обозначается одним или несколькими символами, то эти символы должны быть символами управления форматом, отличными от символа горизонтальной табуляции. Во всяком случае, последовательность из одного или нескольких символов управления форматом, отличных от символа горизонтальной табуляции, должна означать по крайней мере один конец строчки.

Один или несколько разделителей допустимы между любыми двумя соседними лексемами, перед первой или после последней лексемами каждой компиляции. По крайней мере один разделитель необходим между идентификатором или числовым литералом и соседними идентификаторами или числовыми литералами.

Ограничитель – это один из следующих специальных символов из набора основных символов

& ' ( ) * + , – . / : ; < = > |

или один из так называемых *составных ограничителей*, представляющих собой пару специальных символов

=> .. ** := /= >= <= << >> <>

Каждый специальный символ является простым ограничителем, за исключением тех случаев, когда он встречается в составном ограничителе, в комментарии, в строковом, символьном или числовом литералах.

Остальные формы лексем описаны в других разделах этой главы.

Примечание. Каждая лексема должна располагаться в одной строчке, поскольку конец строчки – разделитель. Символы кавычки, номера, подчеркивания и два соседних дефиса не являются ограничителями, но могут входить в лексемы в качестве ее частей.

Наименования составных ограничителей даны в табл. 2.2.

Таблица 2.2

Ограничитель	*Наименование*
=>	Стрелка
. .	Двойная точка
* *	Двойная звездочка (возведение в степень)
: =	Присваивание (читается: „становится равным")
/ =	Неравенство (читается: „не равно")
> =	Больше или равно
< =	Меньше или равно
< <	Левая скобка метки
>>	Правая скобка метки
< >	Бокс (коробка)

2.3. **Идентификаторы**

Идентификаторы используются в качестве имен и зарезервированных слов.

идентификатор : : = буква {[подчеркивание] буква_или_цифра}

буква_или_цифра : : = буква | цифра

буква : : = прописная_буква | строчная_буква

Все символы идентификатора существенны, включая символ подчеркивания между соседними буквами или цифрами. Идентификаторы, различающиеся только размерами букв и использованием графически совпадающих букв русского и латинского алфавитов, считаются совпадающими.

Примеры:

СЧЕТЧИК X дай_символ Эвелина Марион

СНОБОЛ_4 XI СчетчикСтраниц ЗАПАСТИ_СЛЕДУЮЩИЙ_ЭЛЕМЕНТ

Примечание. Пробел внутри идентификатора недопустим, поскольку он является разделителем.

2.4. **Числовые литералы**

Числовые литералы подразделяются на два класса: вещественные и целые. Вещественный литерал – это числовой литерал, который включает точку; целый литерал – это числовой литерал без точки. Вещественные литералы являются литералами типа *универсальный_вещественный.* Целые литералы – литералы типа *универсальный целый.*

числовой_литерал : : = десятичный_литерал | литерал_с_основанием

2.4.1. Десятичные литералы

Десятичный литерал – это числовой литерал, выраженный в общепринятой десятичной системе (основание по умолчанию равно десяти).

десятичный_литерал : : = целое [. целое] [порядок]

целое : : = цифра {[подчеркивание] цифра}

порядок : : = E [+] целое | E – целое

Символ подчеркивания между соседними цифрами десятичного литерала не влияет на значение числового литерала. Буква E в порядке может быть строчной или прописной буквой с одним и тем же назначением.

Для получения значения десятичного литерала с порядком следует умножить значение десятичного литерала без порядка на степень десяти, заданную порядком. Порядок для целого литерала не должен содержать знак минус.

Примеры:

12	0	1E6	123_456	- - целые литералы
12.0	0.0	0.456	3.14159_26	- - вещественные литералы
1.34E–12		1.0E+6		- - вещественные литералы с порядком

Примечание. Ведущие нули допускаются. Пробел в числовом литерале недопустим даже между составными частями порядка, поскольку пробел является разделителем. Нулевой порядок для целого литерала допустим.

2.4.2. Литералы с основанием

Литерал с основанием – это числовой литерал, в котором явно указано основание. Основание должно принимать значение от двух до шестнадцати.

литерал_с_основанием : : = основание # целое_с_основанием
[. целое_с_основанием] # [порядок]
основание : : = целое
целое_с_основанием : : =
расширенная_цифра {[подчеркивание] расширенная_цифра}
расширенная_цифра : : = цифра | буква

Символ подчеркивания, заключенный между соседними цифрами литерала с основанием, не влияет на значение этого числового литерала. Основание и порядок, если они есть, записываются в десятичной системе. В качестве расширенных цифр от десяти до пятнадцати допускаются только латинские буквы от A до F. Буква в литерале с основанием (расширенная цифра или буква E в порядке) может быть строчной или прописной с одним и тем же смыслом.

Предполагается обычный смысл обозначения литерала с основанием; в частности, значение каждой расширенной цифры литерала с основанием должно быть меньше основания. Для получения значения литерала с основанием и порядком следует умножить значение литерала с основанием без порядка на основание, возведенное в указанную порядком степень.

Примеры:

```
2#1111_1111#        16#FF#        016#0FF#
- - целые литералы со значением 255
16#E#E1          2#1110_0000#
- - целые литералы со значением 224
16#F.FF#E+2          2#1.1111_1111_111#E11
- - вещественные литералы со значением 4095.0
```

2.5. **Символьные литералы**

Символьный литерал – это какой-либо из 161 графических символов (включая пробел), заключенный между двумя символами апострофа. Символьный литерал имеет значение некоторого символьного типа.

символьный_литерал : : = 'графический_символ

Примеры:

```
'A'  '*'  '''   ''
```

2.6. **Строковые литералы**

Строковый литерал образуется из последовательности (возможно, пустой) графических символов, заключенной между двумя символами кавычки – *строковыми скобками.*

строковый литерал : : = " {графический_символ}"

Строковый литерал имеет значение, являющееся последовательностью значений символов, соответствующих графическим символам строкового литерала, кроме внешних символов кавычки. Для представления кавычки в последовательности значений символов необходимо в соответствующем месте внутри строкового литерала поместить пару соседних символов кавычки. (Это означает, что строковый литерал, включающий два соседних символа кавычки, никогда не рассматривается как два строковых литерала.)

Длина строкового литерала – это количество значений символьного типа в последовательности, его представляющей. (Каждые два соседних символа кавычки в строке считаются одним символом.)

Примеры:

```
" Дневное сообщение"
""                          -- пустой строковый литерал
" "        "A" """"""      -- три строковых литерала длиной 1
"Символы, такие, как }{, % и } допустимы в строковых литералах".
```

Примечание. Строковый литерал должен помещаться на одной строчке, поскольку он является лексемой (см. 2.2). Более длинные последовательности значений графических символов могут быть получены катенацией строковых литералов. Равным образом, катенация констант, описанных в пакетах ASCII и ГОСТ (см. приложение 3), может быть использована для получения последовательности значений символьного типа, которая включает значения неграфических символов (так называемых управляющих символов). Ниже даны примеры использования катенации:

```
"ПЕРВАЯ ЧАСТЬ ПОСЛЕДОВАТЕЛЬНОСТИ СИМВОЛОВ," &
"КОТОРАЯ ПРОДОЛЖАЕТСЯ НА СЛЕДУЮЩЕЙ СТРОЧКЕ"
"последовательность, которая включает" & ГОСТ. ДА & "управляющий символ"
```

2.7. **Комментарии**

Комментарий начинается с двух соседних дефисов и продолжается до конца строчки. Комментарий может помещаться в любой строчке программы. Присутствие или отсутствие комментария не влияет ни на правильность, ни на неправильность программы. Более того, комментарии не влияют на результат программы; их единственное назначение – сделать программу более понятной.

Примеры:

```
-- последнее предложение выражает ту же мысль, что и
-- в языке Алгол 68
end; -- обработка СТРОЧКИ завершена
-- длинный комментарий может быть разбит на две или
-- несколько последовательных строчек
------- первые два дефиса начинают комментарий
```

Примечание. Горизонтальная табуляция может быть использована в комментариях после двух дефисов; это эквивалентно одному или нескольким пробелам (см. 2.2).

2.8. **Прагмы**

Прагма используется для задания информации компилятору. Прагма начинается зарезервированным словом **pragma**, за которым следует идентификатор – имя прагмы.

прагма : : = **pragma** идентификатор
  [(сопоставление_аргумента {, сопоставление_аргумента})] ;
сопоставление_аргумента : : =
  [идентификатор_*аргумента* =>] имя
  | [идентификатор_*аргумента* =>] выражение

Прагмы допустимы только в следующих местах программы:

• После ограничителя точки с запятой, но не внутри раздела формальных параметров или раздела дискриминантов;

• В любом месте, где синтаксические правила допускают синтаксические понятия, в названии которых содержатся слова „описание", „оператор", „спецификатор" или „альтернатива", или одно из синтаксических понятий: вариант и обработчик исключений, но не вместо соответствующих конструкций, а также в любом месте, где допустим компилируемый модуль.

На месторасположение отдельных прагм могут накладываться дополнительные ограничения.

Некоторые прагмы имеют аргументы. Сопоставления аргументов могут быть либо позиционными, либо именованными, как и при сопоставлении параметров в вызовах подпрограмм (см. 6.4). Однако именованные сопоставления возможны, если определены идентификаторы аргументов. Именем аргумента должно быть либо имя, видимое в месте употребления прагмы, либо идентификатор, специфический для этой прагмы.

Предопределенные прагмы описаны в обязательном приложении 2; они должны поддерживаться в каждой реализации. Реализация может определить дополнительные прагмы. Они должны быть описаны в соответствии с обязательным приложением 4. Реализация не должна допускать определение прагм, наличие или отсутствие которых влияет на правильность текста программы. Следовательно, правильность программы не зависит от наличия или отсутствия прагм, определенных реализацией.

Прагма, не определенная в языке, не дает никакого результата, если ее идентификатор не распознан (данной) реализацией. Более того, прагма (как определенная в языке, так и определенная реализацией) не дает никакого результата, если ее размещение или ее аргументы не соответствуют допустимым для этой прагмы. Участок текста, на который распространяется действие прагмы, зависит от прагмы.

Примеры:

```
pragma LIST (OFF),
pragma OPTIMIZE (TIME),
pragma INLINE (УСТАНОВИТЬМАСКУ);
pragma SUPPRESS (RANGE_CHECK, ON => ИНДЕКС);
```

Примечание. Рекомендуется (но не требуется), чтобы реализация выдавала предупреждающие сообщения о прагмах, которые не распознаны, и поэтому проигнорированы.

2.9. Зарезервированные слова

Перечисленные в табл. 2.3 идентификаторы называются *зарезервированными словами.* Они зарезервированы в языке для специальных целей и в стандарте изображаются строчными буквами полужирным шрифтом (после тире приведено наименование на русском языке).

Таблица 2.3

Зарезервированные слова	*Зарезервированные слова*
abort – прекращение	**access** – ссылка_на
abs – абсолютная величина	**all** – все
accept – принятие	**and** –и

Продолжение табл. 2.3

Зарезервированные слова	*Зарезервированные слова*
array – массив	**not** – не
at – положение	**null** – пусто
begin – начало	**of** – из
body – тело	**or** – или
case – выбор	**others** – другие
constant – константа	**out** – выходной
declare – описание	**package** – пакет
delay – задержка	**pragma** – прагма
delta – дельта	**private** – личный
digits – цифр	**procedure** – процедура
do – выполнение	**raise** – возбуждение
else – иначе	**range** – диапазон
elsif – инесли	**record** – запись
end – конец	**rem** – остаток
entry – вход	**renames** – синоним
exception – исключение	**return** – возврат
exit – выход	**reverse** – в_обратном_порядке
for – для	**select** – отбор
function – функция	**separate** – отдельно
generic – настройка	**subtype** – подтип
goto – переход_на	**task** – задача
if – если	**terminate** – завершение
in – в	**then** – то
in – входной	**type** – тип
is – есть	**use** – использовать
limited – лимитируемый	**when** – когда
loop – цикл	**while** – пока
mod – по_модулю	**with** – совместно_с
new – новый	**xor** – либо

Зарезервированные слова не должны использоваться в качестве описываемого идентификатора.

Примечание. Зарезервированные слова, отличающиеся только соответствующими строчными или прописными буквами, рассматриваются как одинаковые (см. 2.3). Идентификаторы некоторых атрибутов, стоящие после апострофа, совпадают с зарезервированными словами (DELTA, DIGITS и RANGE).

2.10. **Допустимые замены символов**

Для основных символов вертикальной черты, номера и кавычки допустимы следующие замены:

• символ вертикальной черты (|) можно заменить восклицательным знаком (!) там, где он используется как ограничитель;

• символ номера (#) в литерале с основанием можно заменить двоеточием (:) при условии, что в этом литерале заменяются оба символа номера;

• символ кавычки (”), использованный как строковая скобка на обоих концах строкового литерала, можно заменить символом процента (%), если последовательность символов строкового литерала не содержит символа кавычки и если в этом литерале заменяются обе строковые скобки. Для представления символа процента внутри последовательности символов строкового литерала должна использоваться пара соседних символов процента, которые рассматриваются как один символ процента.

Эти замены не изменяют смысла программы.

Примечание. Рекомендуется, чтобы замена символов вертикальной черты, номера и кавычки была ограничена случаями, когда соответствующих графических символов нет на терминалах. Заметим, что вертикальная черта изображается на некоторых устройствах прерывистой линией, замена в этом случае не рекомендуется.

Правила для идентификаторов и числовых литералов таковы, что строчные и прописные буквы эквивалентны; эти лексемы могут быть записаны только символами основного набора. Если строковый литерал предопределенного типа STRING содержит символы не из основного набора, та же самая последовательность значений символов может быть получена катенацией строковых литералов, содержащих символы основного набора, и символьных констант, описанных в предопределенных пакетах ASCII и ГОСТ. Таким образом, строковый литерал ”АБ ¤ ВГ” можно заменить на ”АБ” & ГОСТ.ДЕНЕЖНЫЙ_ЗНАК&”ВГ”, а строковый литерал ”ABcd” со строчными буквами можно заменить на ”AB”&ASCII. LC_C&ASCII.LC_D.

3. ОПИСАНИЯ И ТИПЫ

Эта глава описывает типы и правила описания констант, переменных и именованных чисел.

3.1. **Описания**

В языке определены некоторые сорта понятий, объявляемые явно и неявно описаниями.

К таким понятиям относятся, например, числовой литерал, объект, дискриминант, компонент записи, параметр цикла, исключение, тип, подтип, подпрограмма, пакет, задачный модуль, настраиваемый модуль, одиночный вход, семейство входов, формальный параметр (подпрограммы, входа, настраиваемой подпрограммы), формальный параметр настройки, именованный блок или цикл, помеченный оператор, а также операция (в частности, атрибут или литерал перечисления, см. 3.3.3).

Существует несколько форм описаний. Основное описание – это форма описания, определенная следующим образом:

основное_описание : : =

описание_объекта	| описание_числа
| описание_типа	| описание_подтипа
| описание_подпрограммы	| описание_пакета
| описание_задачи	| описание_настройки

| описание_исключения | конкретизация_настройки
| описание_переименования | описание_субконстанты

Некоторые формы описания всегда являются (явно) частью основного описания, а именно: спецификация дискриминантов, описание компонентов, описание входов, спецификация параметров, описание параметров настройки и спецификации литералов перечисления. Спецификация параметра цикла – это конструкция, которая входит только в некоторые формы оператора цикла.

Остальные формы описания являются неявными: имя блока, имя цикла и метка оператора. Некоторые операции описываются неявно (см. 3.3.3).

Для каждой формы описания правила языка определяют некоторый участок текста, называемый *областью действия* описания (см. 8.2). Некоторые формы описания связывают с описанным понятием идентификатор. Внутри области действия, и только в ней, существуют места, где возможно использование идентификатора для связи его с описанным понятием; эти места определяются правилами видимости (см. 8.3). В таких местах идентификатор называется *именем* понятия (простым именем); говорят, что имя *обозначает* связанное с ним понятие.

Некоторые формы спецификации литерала перечисления связывают символьный литерал с соответствующим описываемым понятием. Некоторые формы описаний связывают символ операции или другие обозначения с явно или неявно описанной операцией.

Процесс, в результате которого описание вступает в силу, называется *предвыполнением* описания; этот процесс протекает во время выполнения программы.

Говорят, что после своего предвыполнения описание становится *предвыполненным.* До завершения своего предвыполнения (как и до его начала) описание считается еще не предвыполненным. Предвыполнение любого описания дает всегда по крайней мере один эффект: описание из еще не предвыполненного становится (меняет статус) предвыполненным. Фраза: *„предвыполнение не дает другого эффекта"* используется тогда, когда происходит только изменение статуса описания. Процесс предвыполнения определяется также для разделов описаний, элементов описания и компилируемых модулей (см. 3.9 и 10.5).

Описания объекта, числа, типа и подтипа даны в этой главе. Остальные основные описания изложены в последующих главах.

Примечание. Синтаксические правила используют термин *идентификатор* для первого вхождения идентификатора в некоторую форму описания; термин *простое имя* используется для любого вхождения идентификатора, который уже обозначает какое-либо описанное понятие.

3.2. **Объекты и именованные числа**

Объект – это понятие языка. Объект имеет (содержит) значение данного типа. Объектом может быть:

• Объект, объявленный описанием объекта или описанием одиночной задачи;

· Формальный параметр подпрограммы, входа или настраиваемой подпрограммы;

· Формальный объект настройки;

· Параметр цикла;

· Объект, указанный значением ссылочного типа;

· Компонент или отрезок другого объекта.

Описание числа – это специальная форма описания объекта, которая связывает идентификатор со значением типа *универсальный_целый* или *универсальный_вещественный.*

описание_объекта : : =
  список_идентификаторов : [**constant**]
    указание_подтипа [: = выражение] ;
| список идентификаторов : [**constant**]
    определение_ограниченного_индексируемого_типа
    [: = выражение] ;
описание _числа : : =
  список_идентификаторов : **constant**
    : = *универсальное_статическое_выражение;*
список_идентификаторов : : =
  идентификатор {, идентификатор}

Описание объекта называется *единичным описанием* объекта, если список его идентификаторов имеет единственный идентификатор; оно называется *групповым описанием объектов*, если его список имеет два или несколько идентификаторов. Групповое описание объектов эквивалентно последовательности соответствующего числа единичных описаний объектов. Для каждого идентификатора из списка в такой эквивалентной последовательности единичное описание объекта формируется из идентификатора, двоеточия и всего того, что стоит справа от двоеточия в групповом описании объекта; описания в эквивалентной последовательности идут в том же порядке, что и список идентификаторов.

Аналогичная эквивалентность имеет место также для списка идентификаторов описания числа, описаний компонентов, спецификаций дискриминантов, спецификаций параметров и описаний параметров настройки, исключений и субконстант.

В остальной части описания языка все пояснения даны для описаний с единственным идентификатором; соответствующие пояснения для описаний с несколькими идентификаторами следуют из эквивалентности, установленной выше.

Примеры:

```
-- групповое описание объектов
ИВАН, ПЕТР: ИМЯ_ПЕРСОНЫ : = new ПЕРСОНА (ПОЛ => М);
-- см. 3.8.1
-- эквивалентно единичным описаниям объектов,
-- следующим в данном порядке
ИВАН: ИМЯ_ПЕРСОНЫ: = new ПЕРСОНА (ПОЛ => М);
ПЕТР: ИМЯ_ПЕРСОНЫ: = new ПЕРСОНА (ПОЛ => М);
```

3.2.1. Описания объектов

Описание объекта вводит объект, тип которого задан либо указанием подтипа, либо определением ограниченного индексируемого типа. Если описание объекта включает составной ограничитель присваивания, за которым следует выражение, то это выражение определяет начальное значение описываемого объекта; тип выражения должен совпадать с типом объекта.

Описываемый объект – *константа,* если в описании объекта присутствует зарезервированное слово **constant**. В этом случае описание должно включать явную инициализацию. Значение константы не может быть изменено после инициализации. Формальные параметры вида **in** подпрограмм и входов, а также формальные параметры настройки вида **in** являются константами; параметр цикла – константа в соответствующем цикле; подкомпонент или отрезок константы – тоже константа.

Объект, не являющийся константой, называется *переменной* (в частности, объект, заданный описанием объекта без зарезервированного слова **constant**, является переменной). Для изменения значения переменной существует только два пути: непосредственное присваивание или косвенное изменение (см. 6.2) оператором вызова процедуры или входа (это действие может быть выполнено над самой переменной, над компонентом переменной, либо над другой переменной, для которой данная является подкомпонентом).

Предвыполнение описания объекта происходит следующим образом:

а) устанавливается подтип объекта посредством предвыполнения указания подтипа или определения ограниченного индексируемого типа;

б) если описание объекта включает явную инициализацию, то его начальное значение получается вычислением соответствующего выражения. В противном случае вычисляются неявные начальные значения (если они есть) объекта или его подкомпонентов;

в) создается объект;

г) начальное значение (заданное явно или по умолчанию) присваивается объекту или соответствующему подкомпоненту.

Неявные начальные значения определяются для объектов, заданных описанием объекта, и для компонентов таких объектов в следующих случаях:

• Для объекта ссылочного типа – его неявное начальное значение равно пустому значению ссылочного типа;

• Для объекта заданного типа – неявное начальное (и единственное) значение обозначает соответствующую задачу;

. Если тип объекта является типом с дискриминантами и его подтип ограничен, то неявное начальное (и единственное) значение каждого дискриминанта определяется подтипом объекта;

• Для объекта составного типа неявное начальное значение каждого компонента, имеющего выражение по умолчанию, получается вычислением этого выражения, если только компонент не дискриминант ограниченного объекта (предыдущий случай).

Если компонент сам является составным объектом, значение которого не определено ни явной инициализацией, ни выражением по умолчанию, то неявное начальное значение компонентов составного объекта определяется теми же самыми правилами, что и для описанного объекта.

Шаги от *а* до *г* выполняются в указанном порядке. Если на шаге *б* вычисляется выражение по умолчанию для дискриминанта, то это вычисление выполняется до вычисления выражений по умолчанию для зависящих от дискриминанта подкомпонентов, а также до вычисления выражений по умолчанию, содержащих имя дискриминанта. Кроме предыдущего правила, порядок вычисления выражений по умолчанию языком не определен.

При инициализации описанного объекта или одного из его подкомпонентов проверяется принадлежность начального значения подтипу объекта; для массива, объявленного описанием объекта, сначала применяется неявное преобразование подтипа, как при выполнении оператора присваивания, если только объект не является константой с подтипом неограниченного индексируемого типа. При отрицательном результате проверки возбуждается исключение CONSTRAINT_ERROR.

Значение скалярной переменной после предвыполнения соответствующего описания объекта не определено, если начальное значение не было присвоено переменной при (явной или неявной) инициализации.

Если операнд преобразования типа или квалифицированного выражения является составной переменной с неопределенными значениями скалярных подкомпонентов, то значения соответствующих подкомпонентов результата неопределены. Выполнение программы ошибочно, если делается попытка вычислить скалярную переменную с неопределенным значением. Аналогично выполнение программы ошибочно, если делается попытка применить предопределенную операцию к составной переменной, имеющей скалярный подкомпонент с неопределенным значением.

Примеры описаний переменных:

```
СЧЕТ, СУММА: INTEGER;
РАЗМЕР: INTEGER  range  0 . . 10_000 : = 0;
СОРТИРОВАН: BOOLEAN : = FALSE;
ТАБЛИЦА_ЦВЕТОВ: array (1 . . K) of ЦВЕТ;
РЕЖИМ: ВЕКТОР_БИТ (1 . . 10) : = (others => TRUE);
```

Примеры описаний констант:

```
ПРЕДЕЛ          : constant INTEGER : = 10_000;
НИЖНИЙ_ПРЕДЕЛ: constant INTEGER : = ПРЕДЕЛ/10;
ДОПУСК          : constant ВЕЩЕСТВ : = ДИСПЕРСИЯ (1.15);
```

Примечание. Выражение для инициализации константы не обязательно является статическим выражением (см. 4.9). В приведенных выше примерах ПРЕДЕЛ и НИЖНИЙ_ПРЕДЕЛ инициализированы статическими выражениями, а ДОПУСК – нет, если ДИСПЕРСИЯ – определенная пользователем функция.

3.2.2. Описание чисел

Описание числа – это специальная дополнительная форма описания константы. Тип статического выражения, заданного для инициализации в описании числа, должен быть либо типом *универсальный_целый*, либо типом

универсальный_вещественный. Константа, объявленная описанием числа, называется *именованным числом* и имеет тот же тип, что и статическое выражение.

Примечание. Относящиеся к выражениям универсального типа правила изложены в разд. 4.10. Из этих правил следует, что именованное число имеет *универсальный_целый* тип, если каждое содержащееся в выражении первичное имеет этот тип. Аналогично, если каждое первичное имеет тип *универсальный_вещественный*, то именованное число имеет этот тип.

Примеры описаний чисел:

```
ПИ              : constant : = 3.14159_26536; - - вещественное
                             - - число
ДВА_ПИ          : constant : = 2.0*ПИ; - - вещественное число
МАКСИМУМ        : constant : = 500;      - - целое число
СТЕПЕНЬ_16      : constant : = 2 * * 16; - - целое 65_536
ОДИН, ONE, EINS : constant : = 1;        - - три различных имени 1
```

3.3. Типы и подтипы

Тип характеризуется набором значений и набором операций (точнее: операций типа или операций над типом).

Существует несколько *классов* типов. *Скалярные* типы – это целые и вещественные типы и типы, определенные перечислением своих значений; значения этих типов не имеют компонентов. *Индексируемый* и *именуемый* типы являются составными. Значение составного типа состоит из значений *компонентов*. *Ссылочный* тип – это тип, значения которого обеспечивают доступ к объектам. *Личные* типы – это типы, для которых полностью определяется набор возможных значений, но непосредственный доступ к ним пользователей невозможен. Наконец, существуют *задачные* типы. (Личные типы описаны в гл. 7, задачные – в гл. 9, остальные – в гл. 3).

Именуемые и личные типы могут иметь специальные компоненты, называемые *дискриминантами,* значения которых различают альтернативные формы значений каждого из этих типов. Если личный тип имеет дискриминанты, они известны пользователям типа. Следовательно, личный тип известен только своим именем, своими дискриминантами, если они есть, и соответствующим набором операций.

Набор возможных значений данного типа может зависеть от условия, которое называется *ограничением* (сюда же относятся случаи без ограничения); значение *удовлетворяет* ограничению, если оно удовлетворяет соответствующему условию. *Подтип* – это тип вместе с ограничением; говорят, что значение *принадлежит подтипу,* если оно принадлежит типу и удовлетворяет ограничению; данный тип называется *базовым типом* подтипа. Тип является подтипом самого себя; такой подтип называется *неограниченным;* он соответствует условию, которое не налагает никаких ограничений. Базовым типом типа является он сам.

Набор операций, определенных над конкретным типом, определен и для любого его подтипа; однако переменной данного подтипа можно присвоить значение только этого подтипа. Дополнительные операции, например, квалификация (в квалифицированном выражении), неявно определяются описанием подтипа.

Для объектов некоторых типов определено *начальное значение по умолчанию;* некоторые другие типы имеют *выражения по умолчанию,* определенные для части или всех своих компонентов. Некоторые операции над типами и подтипами называются *атрибутами;* эти операции обозначаются формой имени, описанной в разд. 4.1.4.

Термин *подкомпонент* используется в описании языка вместо термина компонент, чтобы указать компонент другого компонента или подкомпонента. Если нет других подкомпонентов, используется термин компонент.

Данный тип не должен иметь подкомпонентов, типом которых является он сам.

Имя класса типов используется в описании языка для квалификации объектов и значений, принадлежащих к типу рассматриваемого класса. Например, термин „индексируемый объект" используется для объекта индексируемого типа; аналогично термин „ссылочное значение" используется для значения ссылочного типа.

Примечание. Набор значений подтипа – это подмножество значений базового типа. Это подмножество не обязано быть собственным подмножеством; оно может быть пустым.

3.3.1. Описания типов

Описание типа объявляет тип.

```
описание_типа : : = полное_описание_типа
  | неполное_описание_типа | описание _личного_типа
полное_описание_типа : : =
  type идентификатор  [раздел_дискриминантов]
        is определение_типа;
определение_типа : : = определение_перечислимого_типа
  | определение_целого_типа
  | определение_вещественного_типа
  | определение_индексируемого_типа
  | определение_именуемого_типа
  | определение_ссылочного_типа
  | определение_производного_типа
```

Предвыполнение полного описания типа состоит из предвыполнения раздела дискриминантов, если он есть (исключая случай, когда дается полное описание типа для уже встречавшегося неполного описания типа или для описания личного типа), и предвыполнения определения типа.

Типы, созданные в результате предвыполнения различных определений, являются различными. Более того, предвыполнение определения типа для числовых или производных типов создает как базовый тип, так и подтип базового типа; то же самое следует сказать об определении ограниченного индексируемого типа (одной из двух форм определения индексируемого типа).

Простое имя в полном описании типа обозначает описанный тип, если только описание типа не объявляет одновременно базовый тип и подтип базового типа; в этом случае простое имя обозначает подтип, а базовый тип

является анонимным. Тип называется *анонимным,* если он не имеет простого имени. Для наглядности в этом стандарте время от времени используется псевдоимя анонимного типа, выделенное курсивом там, где обычно по синтаксису требуется идентификатор.

Примеры определений типов:

```
(БЕЛЫЙ, КРАСНЫЙ, ЖЕЛТЫЙ, ЗЕЛЕНЫЙ, ГОЛУБОЙ, КОРИЧНЕВЫЙ, ЧЕРНЫЙ)
range 1 . . 72
array (1 . . 10) of INTEGER
```

Примеры описаний типов:

```
type ЦВЕТ is (БЕЛЫЙ, КРАСНЫЙ, ЖЕЛТЫЙ, ЗЕЛЕНЫЙ,
              ГОЛУБОЙ, КОРИЧНЕВЫЙ, ЧЕРНЫЙ);
type СТОЛБЕЦ is range 1 . . 72;
type ТАБЛИЦА is array (1 . . 10) of INTEGER;
```

Примечание. Два определения типа всегда определяют два различных типа, даже если они текстуально идентичны. Таким образом, данные ниже описания A и B задают различные индексируемые типы:

```
A: array (1 . . 10) of BOOLEAN;
B: array (1 . . 10) of BOOLEAN;
```

Если A и B описаны в групповом описании объектов

```
A,B: array (1 . . 10) of BOOLEAN;
```

то их типы (анонимные) тем не менее различны, так как это групповое описание объектов эквивалентно двум приведенным выше единичным описаниям.

Неполные описания типов используются для определения рекурсивных и взаимосвязанных типов (см. 3.8.1). Описания личных типов используются в спецификациях пакетов и в описаниях параметров настройки (см. 7.4 и 12.1).

3.3.2. Описание подтипов

Описание подтипа объявляет подтип.

```
описание_подтипа : : =
  subtype идентификатор is указание_подтипа;
указание_подтипа : : = обозначение_типа [ограничение]
обозначение_типа : : = имя_типа | имя_подтипа
ограничение : : = ограничение_диапазона
  | ограничение_плавающего_типа
  | ограничение_фиксированного_типа
  | ограничение_индекса
  | ограничение_дискриминанта
```

Обозначение типа обозначает тип или подтип. Если обозначение типа – имя типа, то оно обозначает этот тип, а также соответствующий неограниченный подтип. *Базовым типом,* соответствующим *обозначению типа,* является, по определению, базовый тип типа или подтипа, указанного обозначением типа.

Указание подтипа определяет подтип базового типа, соответствующего обозначению типа.

Если в указании подтипа после обозначения типа стоит ограничение индекса, то обозначение типа не должно обозначать подтип с уже ограниченным индексом. Аналогично, для ограничения дискриминанта обозначение типа не должно иметь ограничение дискриминанта.

Предвыполнение описания подтипа состоит из предвыполнения указания подтипа. Это предвыполнение создает подтип. Если указание подтипа не включает ограничение, то определяемый подтип тот же, что и указанный обозначением типа. Предвыполнение указания подтипа, содержащего ограничение, происходит следующим образом:

а) вначале предвыполняется ограничение;

б) ограничение проверяется на *совместимость* с типом или подтипом, заданным обозначением типа.

После предвыполнения ограничения получается условие, наложенное ограничением. (Правила предвыполнения ограничения таковы, что выражения и диапазоны ограничений вычисляются при предвыполнении всех этих ограничений.) Правила определения совместимости даны в соответствующих разделах для каждой формы ограничения. Эти правила таковы, что если ограничение совместимо с подтипом, то наложенное ограничением условие не может противоречить никакому условию, уже заданному для значений этого подтипа. В противном случае возбуждается исключение CONSTRAINT_ERROR.

Примеры описаний подтипов:

```
subtype РАДУГА is ЦВЕТ range КРАСНЫЙ . . ГОЛУБОЙ; - - см. 3.3.1
subtype КРАСНЫЙ_голубой is РАДУГА;
subtype ЦЕЛ is INTEGER;
subtype МАЛОЕ_ЦЕЛ is INTEGER range - - 10 . . 10;
subtype ВПЛОТЬ_ДО_К is СТОЛБЕЦ range 1 . . К; - - см. 3.3.1
subtype КВАДРАТ is МАТРИЦА (1 . . 10, 1 . . 10); - - см. 3.6
subtype МУЖЧИНА is ПЕРСОНА (ПОЛ => М);      - - см. 3.8
```

Примечание. Описание подтипа не определяет нового типа.

3.3.3. Классификация операций

Набор операций над типом включает явно описанные подпрограммы с параметром или результатом этого типа; такие подпрограммы необходимо описывать после описания типа.

Остальные операции неявно описываются сразу после каждого описания соответствующего типа. К ним относятся *базовые* операции, предопределенные операции (см. 4.5) и литералы перечисления. Описанием производного типа неявно задаются операции, включающие производные подпрограммы. Считается, что неявные описания операций расположены между описанием типа и последующим описанием, если таковое имеется. Неявные описания производных подпрограмм расположены последними.

Базовыми операциями являются:

- Присваивание (в операторах присваивания и инициализациях), генератор, проверка принадлежности и форма управления с промежуточной проверкой;
- Именуемый компонент, индексируемый компонент и отрезок;
- Квалификация (в квалифицированных выражениях), явное преобразование типа и неявное преобразование значения типа *универсальный_целый* или *универсальный_вещественный* в соответствующее значение другого числового типа;

• Числовой литерал (для универсального типа), литерал **null** (для ссылочного типа), строковый литерал, агрегат или атрибут.

Для каждого типа или подтипа Т определен следующий атрибут: T'BASE. Базовый тип Т. Этот атрибут допустим только в качестве префикса имени другого атрибута, например, T'BASE'FIKST.

Примечание. Каждый литерал – это операция, в результате выполнения которой вырабатывается соответствующее значение (см. 4.2). Подобно этому, агрегат – это операция, в результате выполнения которой вырабатывается значение составного типа (см. 4.3). Некоторые операции *оперируют* со значениями данного типа, например, предопределенные операции и некоторые подпрограммы и атрибуты. Некоторые операции *возвращают* значение данного типа, например, литералы и некоторые функции, атрибуты и предопределенные операции. Присваивание – это операция, которая оперирует с объектом и значением. В результате выполнения операции, соответствующей именуемому компоненту, индексируемому компоненту или отрезку, вырабатывается объект или значение, обозначенное этой формой имени.

3.4. Производные типы

Определение производного типа задает новый (базовый) тип, который наследует свойства *родительского типа;* новый тип называется *производным типом.* Определение производного типа создает одновременно *производный подтип,* являющийся подтипом производного типа.

определение_производного_типа : : =
new указание_подтипа

Указание подтипа после зарезервированного слова **new** определяет *родительский подтип.* Родительский тип является базовым для родительского подтипа. Если для родительского подтипа существует ограничение, то подобное ограничение существует и для производного подтипа; разница состоит только в том, что для ограничения диапазона и для ограничения плавающего или фиксированного типов, которое включает ограничение диапазона, значение каждой границы заменяется на соответствующее значение производного типа. Производный тип обладает следующими свойствами:

• Производный тип относится к тому же самому классу типов, что и родительский тип. Набор возможных значений для производного типа есть копия набора возможных значений для родительского типа. Если родительский тип составной, такие же компоненты существуют и у производного типа, а подтип соответствующих компонентов тот же самый;

• Для каждой базовой операции над родительским типом существует соответствующая базовая операция над производным типом. Допускается явное преобразование значения родительского типа в соответствующее значение производного типа и наоборот, как поясняется в разд. 4.6;

• Для каждого литерала перечисления или для каждой предопределенной операции над родительским типом существует соответствующая операция над производным типом;

• Если родительский тип – задачный тип, то для каждого входа родительского типа существует соответствующий вход производного типа;

• Если выражение по умолчанию существует для компонентов объекта, имеющего родительский тип, то то же самое выражение используется для соответствующего компонента объекта производного типа;

• Если родительский тип — ссылочный тип, то родительский и производный типы имеют один и тот же набор значений; существует пустое ссылочное значение для производного типа, которое по умолчанию является начальным значением этого типа;

• Если существует явный спецификатор представления для родительского типа и если этот спецификатор расположен до (но не после) определения производного типа, соответствующий спецификатор представления неявно задан и для производного типа;

• Некоторые подпрограммы, являющиеся операциями над родительским типом, называются *наследуемыми.* Для каждой наследуемой подпрограммы родительского типа имеется соответствующая производная подпрограмма над производным типом. Существуют два сорта наследуемых подпрограмм. Во-первых, если родительский тип описан непосредственно в видимом разделе пакета, то подпрограмма, явно описанная непосредственно в этом видимом разделе, становится наследуемой после конца видимого раздела (если подпрограмма — операция над родительским типом). (Явное описание — это описание подпрограммы, описание переименования подпрограммы или конкретизация настройки.) Во-вторых, если родительский тип сам является производным типом и не описан в видимом разделе пакета, то подпрограмма, которая стала производной, и она не скрыта наследуемой подпрограммой первого сорта, является далее наследуемой.

Описание производного типа неявно описывает в этом месте каждую операцию над производным типом. Неявные описания любых производных подпрограмм следуют за описанием производного типа.

Спецификация производной подпрограммы неявно получается систематическим замещением родительского типа на производный тип в спецификации наследуемой подпрограммы. Любой подтип родительского типа подобным образом замещается подтипом производного типа с аналогичными ограничениями (как при замене ограничения родительского подтипа на соответствующее ограничение производного подтипа). Наконец, любое выражение родительского типа становится операндом преобразования типа, которое вырабатывает результат производного типа.

Вызов производной подпрограммы эквивалентен вызову соответствующей подпрограммы родительского типа, в котором каждый фактический параметр производного типа заменяется преобразованием типа этого фактического параметра к родительскому типу (это означает, что преобразование к родительскому типу происходит перед вызовом параметров вида **in** и **in out**; обратное преобразование к производному типу происходит после вызова параметров вида **in out** и **out**, см. 6.4.1). Дополнительно, если результат вызванной функции имеет родительский тип, то он преобразуется к производному типу.

Если производный тип или личный тип описаны непосредственно в видимом разделе пакета, то в этом разделе этот тип не должен использоваться как родительский тип в определении производного типа (для личных типов см. также разд. 7.4.1).

При предвыполнении определения производного типа сначала предвыполняется указание подтипа, затем создается производный тип, а затем – производный подтип.

Примеры:

```
type ЛОКАЛЬНАЯ_КООРДИНАТА is new КООРДИНАТА; - - два различных типа
type СЕРЕДИНА_НЕДЕЛИ is new ДЕНЬ range ВТР . . ЧТВ; - - см. 3.5.1
type СЧЕТЧИК is new POSITIVE; - - тот же диапазон, что и POSITIVE
type СПЕЦИАЛЬНЫЙ_КЛЮЧ is new УПРАВЛЕНИЕ_ПО_КЛЮЧУ . КЛЮЧ;
                                                    - - см. 7.4.2
- - Производные подпрограммы имеют следующие спецификации:
- - procedure ДАЙ_КЛЮЧ (К: out СПЕЦИАЛЬНЫЙ_КЛЮЧ);
- - function "<" (Х, У: СПЕЦИАЛЬНЫЙ_КЛЮЧ) return BOOLEAN;
```

Примечание. Из правил наследования базовых операций и литералов перечисления следует, что обозначения для литерала или агрегата производного типа те же самые, что и для родительского типа; такие литералы и агрегаты называются *совмещенными*. Из правил также следует, что обозначения компонента, дискриминанта, входа, отрезка или атрибута одинаковы для производного и родительского типов.

Скрытие производной подпрограммы допустимо даже в одной и той же зоне описания (см. 8.3). Производная подпрограмма скрывает предопределенную операцию, имеющую тот же профиль типа параметра и результата (см. 6.6).

Описание настраиваемой подпрограммы не наследуется, поскольку оно описывает настраиваемый модуль, а не подпрограмму. С другой стороны, конкретизация настраиваемой подпрограммы является (ненастраиваемой) подпрограммой, которая является наследуемой, если она удовлетворяет требованиям наследуемости подпрограмм.

Если родительский тип является логическим, то предопределенные операции отношения над производным типом дают результаты предопределенного типа BOOLEAN (см. 4.5.2).

Если спецификатор представления дан для родительского типа и помещен за описанием производного типа, то соответствующий спецификатор представления неприменим к производному типу; следовательно, для такого производного типа допускается явный спецификатор представления.

Если параметр производной подпрограммы принадлежит производному типу, подтип этого параметра не обязан иметь некоторое значение, общее со значением производного подтипа.

3.5. Скалярные типы

Под скалярными типами подразумеваются перечислимые, целые и вещественные типы. Перечислимые и целые типы называются *дискретными* типами; каждое значение дискретного типа имеет номер позиции – целое значение. Целые и вещественные типы называются *числовыми* типами. Все скалярные типы упорядочены, т. е. для их значений предопределены все операции отношения.

ограничение_диапазона : : = **range** диапазон

диапазон : : = атрибут_*диапазона*

| простое_выражение . . простое_выражение

Диапазон определяет подмножество значений скалярного типа. Диапазон Л . . П определяет значения от Л до П включительно, если истинно отношение Л <= П. Значения Л и П называются соответственно *нижней границей* и *верхней границей* диапазона. Значение Х *удовлетворяет* ограничению диапазона, если оно принадлежит диапазону; говорят, что значение Х *принадле-*

жит диапазону, если одновременно истинны отношения Л <= Х и Х <= П. *Пустой диапазон* – это диапазон, для которого истинно отношение П < Л; пустому диапазону не принадлежит никакое значение. Операции <= и < в вышеприведенных определениях являются предопределенными операциями над скалярными типами.

Если ограничение диапазона используется в указании подтипа, либо непосредственно, либо как часть ограничения плавающего или фиксированного типа, тип простых выражений (а также границ атрибута диапазона) должен быть тем же, что и базовый тип обозначения типа указания подтипа. Ограничение диапазона *совместимо* с подтипом, если каждая граница диапазона принадлежит подтипу или если ограничение диапазона определяет пустой диапазон; иначе ограничение диапазона не совместимо с подтипом.

Предвыполнение ограничения диапазона состоит из вычисления диапазона. Вычисление диапазона определяет его нижнюю и верхнюю границы. Если границы заданы простыми выражениями, вычисление диапазона вычисляет эти выражения в порядке, который не определен в языке.

Атрибуты.

Для любого скалярного типа Т или для любого подтипа Т скалярного типа определены следующие атрибуты:

Т'FIRST Вырабатывает значение нижней границы Т. Значение этого атрибута имеет тип Т;

Т'LAST Вырабатывает значение верхней границы Т. Значение этого атрибута имеет тип Т.

Примечание. Индексирование и правила итерации используют значения дискретных типов.

3.5.1. Перечислимые типы

Определение перечислимого типа задает перечислимый тип.

определение_перечислимого_типа : : =
(спецификация_литерала_перечисления
{, спецификация_литерала_перечисления})

спецификация_литерала_перечисления : : =
литерал_перечисления

литерал_перечисления : : =
идентификатор | символьный_литерал

Идентификаторы и символьные литералы, перечисленные в определении перечислимого типа, должны быть различными. Каждая спецификация литерала перечисления является его описанием: это описание эквивалентно описанию функции без параметров, обозначение которой – этот литерал перечисления, а тип результата – определяемый перечислимый тип. Предвыполнение определения перечислимого типа создает перечислимый тип; это предвыполнение включает предвыполнение каждой спецификации литерала перечисления.

Каждый литерал перечисления вырабатывает отличное от других перечислимое значение. Предопределенные операции отношения упорядоченнос-

ти между перечислимыми значениями учитывают порядок, соответствующий номеру позиции; номер позиции первого значения в списке литералов перечисления равен нулю; номер каждого следующего литерала перечисления на единицу больше номера предыдущего литерала в списке.

Если один и тот же идентификатор или символьный литерал задан в нескольких определениях перечислимого типа, соответствующие литералы называются *совмещенными*. В тексте программы тип совмещенного литерала перечисления должен быть определимым из контекста (см. 8.7).

Примеры:

```
type ДЕНЬ is (ПНД, ВТР, СРД, ЧТВ, ПТН, СББ, ВСК);
type МАСТЬ is (ТРЕФЫ, БУБНЫ, ЧЕРВЫ, ПИКИ);
type РОД is (М, Ж);
type УРОВЕНЬ is (НИЗШИЙ, СРЕДНИЙ, СРОЧНЫЙ);
type ЦВЕТ is (БЕЛЫЙ, КРАСНЫЙ, ЖЕЛТЫЙ, ЗЕЛЕНЫЙ, ГОЛУБОЙ,
              КОРИЧНЕВЫЙ, ЧЕРНЫЙ);
type СВЕТ is (КРАСНЫЙ, ЯНТАРНЫЙ, ЗЕЛЕНЫЙ):
                -- КРАСНЫЙ и ЗЕЛЕНЫЙ совмещены.
type ШЕСТНАДЦАТЕРИЧНЫЕ is ('A','B','C', 'D', 'E', 'F');
type СМЕШАННЫЙ is ('A', 'B', '*', B, НИЧТО, '?', '%');
subtype ДЕНЬ НЕДЕЛИ is ДЕНЬ range ПНД . .ПТН;
subtype КОЗЫРЬ is МАСТЬ range ЧЕРВЫ . . ПИКИ;
subtype РАДУГА is ЦВЕТ rangy КРАСНЫЙ . . ГОЛУБОЙ;
                                -- КРАСНЫЙ – цвет, но не свет.
```

Примечание. Если литерал перечисления встречается в контексте, недостаточном для определения типа литерала, то один из путей разрешения неоднозначности – это применение квалифицированного выражения с именем перечислимого типа (см. 8.7).

3.5.2. Символьные типы

Перечислимый тип называется символьным, если хотя бы один из его литералов перечисления является символьным литералом. Предопределенный тип CHARACTER – символьный тип, значения которого представляют собой 195 символов стандартного набора. Каждый из 161 графических символов этого символьного набора обозначен соответствующим символьным литералом.

Пример:

```
type РИМСКАЯ_ЦИФРА is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
```

Примечание. Предопределенные пакеты ASCII и ГОСТ включают описания констант, обозначающих управляющие символы, а также констант, обозначающих графические символы, не входящие в основной набор символов.

Другой символьный набор, например EBCDIC, можно описать как символьный тип; внутренние коды символов можно задать спецификатором представления перечислимых, как пояснено в разд. 13.3.

3.5.3. Логические типы

Существует предопределенный перечислимый тип, именуемый BOOLEAN. Он содержит два литерала FALSE и TRUE, упорядоченные отношением FALSE < TRUE. Логический тип – это тип BOOLEAN или производный, непосредственно или косвенно, от логического типа.

3.5.4. Целые типы

Определение целого типа задает целый тип, набор значений которого включает по крайней мере значения из заданного диапазона.

определение_целого_типа : : = ограничение_диапазона

В ограничении диапазона, используемого в определении целого типа, границы диапазона должны определяться статическими выражениями некоторого, не обязательно одного и того же, целого типа. (Допускаются отрицательные границы.)

Описание типа в форме:

type T **is range** Л . . П;

по определению эквивалентно следующим описаниям:

type *целый_тип* **is new** предопределенный_целый_тип;

subtype T **is** *целый_тип* **range** *целый_тип* (Л) . . *целый_тип* (П):

где *целый_тип* – это анонимный тип, а предопределенный целый тип неявно выбран реализацией и содержит значения от Л до П включительно. Описание целого типа неправильно, если ни один из предопределенных целых типов не удовлетворяет этому требованию, за исключением типа *универсальный_целый*. Предвыполнение описания целого типа состоит из предвыполнения эквивалентных описаний типа и подтипа.

Предопределенные целые типы включают тип INTEGER. Реализация может также иметь предопределенные типы SHORT_INTEGER и LONG_INTEGER, диапазоны которых соответственно (существенно) уже и шире, чем у типа INTEGER. Диапазон для значений этих типов должен быть симметричным относительно нуля, кроме наименьшего из отрицательных значений, которое может существовать в некоторых реализациях. Базовым для каждого из этих типов является он сам.

Целый литерал – это литерал анонимного предопределенного целого типа, который в данном стандарте называется *универсальным_целым*. Другие целые типы не имеют литералов. Однако для каждого целого типа существует неявное преобразование *универсального_целого* значения в соответствующее значение (если оно есть) целого типа. Обстоятельства, в которых применяются такие неявные преобразования, описаны в разд. 4.6.

Номер позиции целого значения – это соответствующее значение типа *универсальный_целый*.

Для всех целых типов предопределены одни и те же арифметические операции (см. 4.5). Исключение NUMERIC_ERROR возбуждается при выполнении операции (в частности, неявного преобразования), которая не может передать корректный результат (т. е. значение, соответствующее математическому результату, не является значением целого типа). Однако от реализации не требуется возбуждения исключения NUMERIC_ERROR, если операция является частью большего выражения, результат которого может быть вычислен корректно, как пояснено в разд. 11.6.

Примеры:

```
type НОМЕР_СТРАНИЦЫ is range 1 . . 2_000;
type РАЗМЕР_СТРОЧКИ is range 1 . . МАКС_РАЗМЕР_СТРОЧКИ;
subtype МАЛОЕ_ЦЕЛ  is INTEGER range    –10 . . 10;
```

```
subtype УКАЗ_СТОЛБЦА is РАЗМЕР_СТРОЧКИ range 1 . . 10;
subtype РАЗМЕР_БУФЕРА is INTEGER range 0 . . МАКС;
```

Примечание. Имя в описании целого числа – это имя подтипа. С другой стороны, предопределенные операции над целым типом определяют результат, который принадлежит диапазону, определяемому родительским предопределенным типом; такой результат необязательно принадлежит описанному подтипу, и попытка присвоить такой результат переменной целого подтипа возбуждает исключение CONSTRAINT_ERROR.

Наименьшее (наибольшее по модулю отрицательное) значение, поддержанное реализацией для предопределенных целых типов, есть именованное число SYSTEM. MIN_INT, а наибольшее (из положительных) значение – SYSTEM.MAX_INT (см. 13.7).

3.5.5. Операции над дискретными типами

Базовые операции над дискретными типами включают присваивание, проверку принадлежности и квалификацию; для логических типов – управление с промежуточной проверкой; для целого типа – явное преобразование значений других числовых типов к этому целому типу и неявное преобразование значений типа *универсальный_целый* к значению заданного типа.

Для каждого дискретного типа или подтипа Т базовые операции включают перечисленные ниже атрибуты. В этом перечислении Т ссылается на подтип (подтип Т) для любого свойства, зависящего от ограничений для Т; другие свойства установлены в терминах базового типа Т.

Первая группа атрибутов вырабатывает характеристики подтипа Т. Эта группа включает атрибут BASE (см. 3.3.3), атрибуты FIRST и LAST (см. 3.5), атрибут представления SIZE (см. 13.7.2) и атрибут WIDTH, определенный следующим образом:

Т'WIDTH Вырабатывает максимальную длину образа по всем значениям подтипа Т (*образ* – это последовательность символов, вырабатываемая атрибутом IMAGE, см. ниже). Вырабатывает нуль для пустого диапазона. Значения этого атрибута имеют тип *универсальный_целый.*

Все атрибуты второй группы – это функции с одним параметром. Соответствующий фактический параметр обозначен ниже идентификатором X.

Т'POS Параметр X должен быть значением базового типа Т. Тип результата – *универсальный_целый.* Результат – номер позиции для значения параметра.

Т'VAL Параметр X может быть любого целого типа. Тип результата – базовый тип Т. По заданному значению X – номеру позиции – функция вырабатывает значение в этой позиции. Если соответствующее X – *универсальное_целое* значение – не принадлежит диапазону Т'POS (Т'BASE 'FIRST) . . Т'POS (Т'BASE' LAST), то возбуждается исключение CONSTRAINT_ERROR.

Т'SUCC Параметр X должен быть значением базового типа Т. Тип результата – базовый тип Т. Результат – значение с номером позиции, на единицу большим номера позиции для значения X. Если X равен Т'BASE'LAST, то возбуждается исключение CONSTRAINT_ERROR.

Т'PRED Параметр X должен быть значением базового типа Т. Тип результата – базовый тип Т. Результат – значение с номером позиции, на единицу меньшим номера позиции для значения X. Если X равно Т'BASE'FIRST, то возбуждается исключение CONSTRAINT_ERROR.

T'IMAGE Параметр X должен быть значением базового типа T. Тип результата – предопределенный тип STRING. Результат – *образ* значения X, т. е. последовательность символов, представляющих изображение значения. Образу целого значения соответствует десятичный литерал без подчеркиваний, предшествующих нулей, порядка и пробелов справа, но с одним символом минус или пробелом слева. Нижняя граница образа есть единица.

Образ литерала перечисления – это либо соответствующий идентификатор из прописных букв, либо соответствующий символьный литерал (включая два апострофа); пробелы не включаются ни слева, ни справа. Образ символа C, отличного от графического символа, зависит от реализации; должно выполняться равенство

C = CHARACTER'VALUE (CHARACTER'IMAGE (C)).

T' VALUE Параметр X должен быть значением предопределенного типа STRING. Тип результата – базовый тип T. Игнорируются любые пробелы слева и справа от последовательности символов, соответствующей параметру.

Если для перечислимого типа последовательность символов имеет синтаксис литерала перечисления и если этот литерал существует для базового типа T, то результат – соответствующее значение перечислимого типа. Если для целого типа последовательность символов имеет синтаксис целого литерала с возможным знаком минус или плюс слева и если существует соответствующее значение базового типа T, то результат есть это значение. Во всех остальных случаях возбуждается исключение CONSTRAINT_ERROR.

Кроме того, для объекта A дискретного типа определены атрибуты A'SIZE и A'ADDRESS (см. 13.7.2).

Кроме базовых, операции над дискретными типами включают предопределенные операции отношения. Для перечислимых типов операции включают литералы перечисления. Для логических типов включают предопределенную унарную логическую операцию отрицания **not** и предопределенные логические операции. Для целых типов операции включают предопределенные *арифметические* операции: унарные и бинарные аддитивные операции – и +, все мультипликативные операции, унарную операцию **abs** и операцию возведения в степень.

Операции над подтипом – это операции над его базовым типом, кроме следующих: присваивания, проверки принадлежности, квалификации, явного преобразования типа и атрибутов первой группы; результат каждой из этих операций зависит от подтипа (присваивание, проверка принадлежности, квалификация и преобразования включают проверку подтипа; атрибуты первой группы вырабатывают характеристику подтипа).

Примечание. Для подтипа дискретного типа переданные атрибутами SUCC, PRED, VAL и VALUE результаты не обязательно принадлежат подтипу; аналогично, фактические параметры атрибутов POS, SUCC, PRED и IMAGE не обязаны принадлежать подтипу. Эти атрибуты удовлетворяют (при отсутствии исключения) следующим соотношениям:

T'POS(T'SUCC (X)) = T'POS (X) + 1

T′ POS (T′PRED (X)) = T′ POS (X) – 1
T′ VAL (T′ POS (X)) = X
T′ POS (T′ VAL (K)) = K

Примеры:

-- для типов и подтипов, описанных в разд. 3.5.1:
-- ЦВЕТ′ FIRST = БЕЛЫЙ, ЦВЕТ′ LAST = ЧЕРНЫЙ,
-- РАДУГА′ FIRST = КРАСНЫЙ, РАДУГА′ LAST = ГОЛУБОЙ,
-- ЦВЕТ′ SUCC (ГОЛУБОЙ) = РАДУГА′ SUCC (ГОЛУБОЙ) = КОРИЧНЕВЫЙ,
-- ЦВЕТ′ POS (ГОЛУБОЙ) = РАДУГА′ POS (ГОЛУБОЙ) = 4,
-- ЦВЕТ′ VAL (0) = РАДУГА′ VAL (0) = БЕЛЫЙ.

3.5.6. Вещественные типы

Вещественные типы обеспечивают приближение вещественных чисел с относительной погрешностью для плавающих типов и с абсолютной погрешностью для фиксированных типов.

определение_вещественного_типа : =
ограничение_плавающего_типа
| ограничение_фиксированного_типа

С каждым вещественным типом связан набор чисел, называемых *модельными числами.* Границы ошибок в предопределенных операциях даны в терминах модельных чисел. Реализация типа должна включать по крайней мере эти модельные числа и представлять их точно.

С каждым вещественным типом также связан зависящий от реализации набор чисел, называемых *хранимыми числами.* Набор хранимых чисел вещественного типа должен включать по крайней мере набор модельных чисел типа. Допустимо, чтобы диапазон хранимых чисел был больше диапазона модельных чисел, но границы ошибок предопределенных операций над хранимыми числами определены теми же правилами, что и для модельных чисел. Хранимые числа по этой причине обеспечивают гарантированные границы ошибок для операций в зависящем от реализации диапазоне чисел; напротив, диапазон модельных чисел зависит только от определения вещественного типа и поэтому не зависит от реализации.

Вещественные литералы – это литералы анонимного предопределенного вещественного типа, называемого в этом стандарте *универсальным_вещественным.* Другие вещественные типы не имеют литералов. Для каждого вещественного типа существует неявное преобразование, которое преобразует *универсальное_вещественное* значение в значение этого вещественного типа. Условия, в которых вызываются эти преобразования, описаны в разд. 4.6. Если *универсальное_вещественное* значение – хранимое число, то неявное преобразование вырабатывает соответствующее значение; если оно принадлежит диапазону хранимых чисел, но не является хранимым числом, то преобразованное значение может быть любым значением в диапазоне, определенном ближайшими хранимыми предыдущим и следующим числами и содержащим данное *универсальное_вещественное* значение.

Выполнение операции вырабатывает значение вещественного типа и может возбудить исключение NUMERIC_ERROR, как поясняется в разд. 4.5.7, если операция не может выработать корректный результат (т. е. соответст-

вующее одному из возможных математических результатов значение не принадлежит диапазону хранимых чисел); в частности, это исключение может быть возбуждено неявным преобразованием. Однако от реализации не требуется возбуждать исключение NUMERIC_ERROR, если операция – часть большего выражения, которое может быть корректно вычислено (см. 11.6).

Предвыполнение определения вещественного типа включает предвыполнение ограничения плавающего или фиксированного типа и создает вещественный тип.

Примечание. Алгоритм, использующий только минимальные свойства чисел, которые гарантированы определением типа для модельных чисел, будет переносимым без каких-либо предосторожностей.

3.5.7. Плавающие типы

Для плавающих типов граница ошибки определяется заданием относительной погрешности в виде требуемого минимального числа значащих десятичных цифр.

ограничение_плавающего_типа : : =
  определение_точности_плавающего_типа
    [ограничение_диапазона]
определение_точности_плавающего_типа : : =
  digits *статическое*_простое_выражение

Минимальное число значащих десятичных цифр определяется значением статического простого выражения в определении точности плавающего типа. Это значение должно быть некоторого целого типа и должно быть положительным (ненулевым); в дальнейшем оно обозначено буквой Д. Если ограничение плавающего типа использовано для определения вещественного типа и включает ограничение диапазона, то каждая граница диапазона должна быть определена статическим выражением некоторого вещественного типа, но две границы не обязаны иметь один и тот же вещественный тип.

Для заданного *основания* определена следующая каноническая форма отличного от нуля модельного числа плавающего типа:

знак * *мантисса* * (*основание* ** *порядок*)

В этой форме: *знак* – либо +1, либо – –1; *мантисса* выражена в системе счисления, заданной *основанием;* порядок – целое число (возможно, отрицательное), такое, что целая часть мантиссы равна нулю, а первая цифра ее дробной части не равна нулю.

Число Д – минимальное требуемое число десятичных цифр после точки в десятичной мантиссе (т. е. если *основание* равно десяти). Значение Д, в свою очередь, определяет соответствующее число В – минимальное требуемое число двоичных цифр после точки в двоичной мантиссе (т. е. если *основание* равно двум). Число В связано с Д и равно такому минимальному значению, что относительная точность двоичной формы не меньше точности для десятичной формы. (Число В равно ближайшему целому, превышающему

$$(Д * \log (10) / \log (2)) + 1.)$$

Модельные числа, заданные определением точности плавающего типа, включают нуль и все числа, у которых двоичная каноническая форма имеет

точно В цифр после точки в мантиссе и порядок в диапазоне –4 * В . . +4 * В. Гарантированная минимальная точность операций над плавающим типом определена в терминах его модельных чисел с ограничением плавающего типа, которое формирует соответствующее определение вещественного типа (см. 4.5.7).

Предопределенные плавающие типы включают тип FLOAT. Реализация может иметь также предопределенные типы SHORT_FLOAT и LONG_FLOAT с точностью, (существенно) меньшей и большей соответственно, чем у FLOAT. Базовым типом каждого предопределенного плавающего типа является он сам. Модельные числа каждого предопределенного плавающего типа определены числом Д десятичных цифр, вырабатываемых атрибутом DIGITS (см. 3.5.8).

Для каждого предопределенного плавающего типа (следовательно, и для каждого производного от него типа) набор хранимых чисел определен следующим образом. Хранимые числа имеют то же самое число цифр В мантиссы, как и модельные числа типа, а порядок в диапазоне – Е . . +Е, где Е зависит от реализации и равно по крайней мере 4 * В для модельных чисел. (Следовательно, хранимые числа включают модельные числа). Правила определения точности операций над модельными и хранимыми числами даны в разд. 4.5.7. Хранимые числа подтипа те же, что и для его базового типа.

Описание плавающего типа, представленное в одной из двух форм (т. е. с возможным ограничением диапазона, обозначенным квадратными скобками, или без него):

type Т **is digits** Д [**range** Л . . П] ;

по определению эквивалентно следующим описаниям:

type *плавающий_тип* **is new**
предопределенный_плавающий_тип;
subtype Т **is** *плавающий_тип* **digits** Д
[**range** *плавающий_тип* (Л) . . *плавающий_тип* (П)] ;

где *плавающий_тип* является анонимным, а предопределенный плавающий тип неявно выбран реализацией так, что его модельные числа включают модельные числа, определенные значением Д; кроме того, если добавлен диапазон Л . . П, то Л и П должны принадлежать диапазону хранимых чисел. Описание плавающего типа неправильно, если ни один из предопределенных плавающих типов не удовлетворяет этим требованиям, кроме типа *универсальный_вещественный.* Максимальное число цифр, которое может быть задано определением точности плавающего типа, определяется именованным числом SYSTEM.MAX_DIGITS (см. 13.7.1).

Предвыполнение описания плавающего типа состоит из предвыполнения эквивалентных описаний типа и подтипа.

Если ограничение плавающего типа следует за обозначением типа в указании подтипа, то обозначение типа должно задавать плавающий тип или подтип. Ограничение плавающего типа *совместимо* с обозначением типа, только если число Д в определении точности плавающего типа не больше соответствующего числа Д для типа или подтипа в обозначении типа. Кроме

того, если ограничение плавающего типа включает ограничение диапазона, то ограничение плавающего типа совместимо с обозначением типа, только если ограничение диапазона совместимо с обозначением типа.

Предвыполнение такого указания подтипа включает предвыполнение ограничения диапазона, если оно есть; оно создает подтип плавающего типа, модельные числа которого определены соответствующим определением точности плавающего типа. Значение плавающего типа принадлежит плавающему подтипу тогда и только тогда, когда оно принадлежит диапазону подтипа.

Одни и те же арифметические операции предопределены для всех плавающих типов (см. 4.5).

Примечание. Ограничение диапазона допустимо в указании плавающего подтипа непосредственно после обозначения типа, либо как часть ограничения плавающего типа. В обоих случаях границы диапазона должны принадлежать базовому типу обозначения типа (см. 3.5). Наложение ограничения плавающего типа на обозначение типа в указании подтипа не может уменьшить допустимый диапазон значений, если оно не включает ограничение диапазона (диапазон модельных чисел, которые соответствуют заданному числу цифр, может быть меньше, чем диапазон чисел обозначения типа). Принадлежащее плавающему подтипу значение не обязательно является модельным числом подтипа.

Примеры:

```
type КОЭФФИЦИЕНТ is digits 10 range –1.0. .1.0;
type ВЕЩЕСТВ is digits 8;
type МАССА is digits      7 range 0.0 . . 1.0E35;
subtype КОРОТКИЙ_КОЭФФ is КОЭФФИЦИЕНТ digits 5;
- - подтип с меньшей точностью
subtype ВЕРОЯТНОСТЬ is ВЕЩЕСТВ range 0.0 . . 1.0;
- - подтип с меньшим диапазоном.
```

Примечания к примерам. Реализованная точность для типа КОЭФФИЦИЕНТ – это точность предопределенного типа, имеющего по меньшей мере 10 цифр мантиссы. Следовательно, спецификация пяти цифр точности для подтипа КОРОТКИЙ_КОЭФФ допустима. Наибольшее модельное число для типа МАССА равно приблизительно 1.27E30 и, следовательно, меньше, чем заданная верхняя граница (1.0E35). Следовательно, описание этого типа правильно, только если эта верхняя граница принадлежит диапазону хранимых чисел предопределенного плавающего типа, имеющего по меньшей мере 7 цифр точности.

3.5.8. Операции над плавающими типами

Базовые операции над плавающим типом включают присваивание, проверку принадлежности, квалификацию, явное преобразование значений других числовых типов в значения этого плавающего типа и неявное преобразование значений типа *универсальный_вещественный* в значения этого типа.

Кроме того, для каждого плавающего типа или подтипа Т базовые операции включают перечисленные ниже атрибуты. В этом перечислении Т ссылается на подтип (подтип Т) для любого свойства, зависящего от наложенных на Т ограничений; другие свойства формулируются через базовый тип Т.

Первая группа атрибутов вырабатывает характеристики подтипа Т. К атрибутам этой группы относятся: атрибут BASE (см. 3.3.3), атрибуты

FIRST и LAST (см. 3.5), атрибут представления SIZE (см. 13.7.2), а также следующие атрибуты:

T'DIGITS Вырабатывает число десятичных цифр в десятичной мантиссе модельных чисел подтипа T. (Этот атрибут вырабатывает число Д, см. 3.5.7.) Значение этого атрибута имеет тип *универсальный_целый.*

T'MANTISSA Вырабатывает число двоичных цифр в двоичной мантиссе модельных чисел подтипа T. (Этот атрибут вырабатывает число В, см. 3.5.7.) Значение этого атрибута имеет тип *универсальный_целый.*

T'EPSILON Вырабатывает абсолютное значение разности между модельным числом 1.0 и следующим модельным числом подтипа T. Значение этого атрибута имеет тип *универсальный_вещественный.*

T'EMAX Вырабатывает наибольшее значение порядка двоичной канонической формы модельных чисел подтипа T. (Этот атрибут вырабатывает произведение 4 * В, см. 3.5.7). Значение этого атрибута имеет тип *универсальный_целый.*

T'SMALL Вырабатывает наименьшее положительное (ненулевое) модельное число подтипа T. Значение этого атрибута имеет тип *универсальный_вещественный.*

T'LARGE Вырабатывает наибольшее положительное модельное число подтипа T. Значение этого атрибута имеет тип *универсальный_вещественный.*

Атрибуты второй группы вырабатывают характеристики хранимых чисел и включают следующие атрибуты:

T'SAFE_EMAX Вырабатывает наибольшее значение порядка двоичной канонической формы хранимых чисел базового типа T. (Этот атрибут вырабатывает число Е, см. 3.5.7). Значение этого атрибута имеет тип *универсальный_целый.*

T'SAFE_SMALL Вырабатывает наименьшее положительное (ненулевое) хранимое число базового типа T. Значение этого атрибута имеет тип *универсальный_вещественный.*

T'SAFE_LARGE Вырабатывает наибольшее положительное хранимое число базового типа T. Значение этого атрибута имеет тип *универсальный_вещественный.*

Кроме этого, для объекта A плавающего типа определены атрибуты A'SIZE и A'ADDRESS (см. 13.7.2). Для каждого плавающего типа существуют машинно-зависимые атрибуты, которые не относятся к модельным и хранимым числам. Они соответствуют обозначениям атрибутов MACHINE_EMAX, MACHINE_EMIN, MACHINE_RADIX, MACHINE_MANTISSA, MACHINE_ROUNDS и MACHINE_OVERFLOWS (см. 13.7.3).

Кроме базовых операций, над плавающим типом определены операции отношения и следующие предопределенные арифметические операции: унарные и бинарные аддитивные операции – и +, мультипликативные операции * и /, унарная операция **abs** и операция возведения в степень.

Операции над подтипом являются соответствующими операциями над типом, кроме следующих: присваивания, проверки принадлежности, квали-

фикации, явного преобразования и атрибутов первой группы; результаты этих операций переопределены в терминах подтипа.

Примечание. Атрибуты EMAX, SMALL, LARGE и EPSILON введены для удобства и связаны с атрибутом MANTISSA следующими формулами:

T'EMAX = 4 * T'MANTISSA

T'EPSILON = 2.0 ** (1 – T'MANTISSA)

T'SMALL = 2.0 ** (–T'EMAX – 1)

T'LARGE = 2.0 ** T'EMAX * (1.0 – 2.0 ** (–T'MANTISSA))

Атрибут MANTISSA, дающий число двоичных цифр в мантиссе, сам связан с атрибутом DIGITS.

Между характеристиками модельных и хранимых чисел справедливы следующие соотношения:

T'BASE'EMAX <= T'SAFE_EMAX

T'BASE'SMALL >= T'SAFE_SMALL

T'BASE'LARGE <= T'SAFE_LARGE

Атрибуты T'FIRST и T'LAST не обязательно вырабатывают модельные или хранимые числа. Если некоторое число цифр определено описанием типа или подтипа T, то атрибут T'DIGITS вырабатывает это число.

3.5.9. Фиксированные типы

Для фиксированных типов граница ошибки определяется абсолютной погрешностью, называемой *дельтой* фиксированного типа.

ограничение_фиксированного_типа : : =
  определение_точности_фиксированного_типа
  [ограничение_диапазона]

определение_точности_фиксированного_типа : : =
  delta *статическое*_простое_выражение

Дельта определяется значением статического простого выражения в определении точности фиксированного типа. Это значение должно принадлежать некоторому вещественному типу и должно быть положительным (ненулевым). Если ограничение фиксированного типа использовано как определение вещественного типа, то оно должно включать ограничение диапазона; каждая граница диапазона должна быть определена статическим выражением некоторого вещественного типа, но эти две границы не обязаны иметь один и тот же тип. Если ограничение фиксированного типа использовано в указании подтипа, то ограничение диапазона необязательно.

Для любого ненулевого модельного числа фиксированного типа определена каноническая форма:

*знак * мантисса * дискрет*

В приводимой форме: *знак* – это либо +1, либо –1; *мантисса* – положительное (ненулевое) целое; любое модельное число кратно некоторому положительному вещественному числу, называемому *дискретом.*

Для модельных чисел фиксированного типа *дискрет* выбран как наибольшая степень двух, которая не превышает дельту определения точности фиксированного типа. *Дискрет* можно задать спецификатором длины (см. 13.2), в этом случае модельные числа кратны заданному значению. Гарантированная минимальная точность операций над фиксированным типом определена в терминах модельных чисел для ограничения фиксированного типа,

которое образует соответствующее определение вещественного типа (см. 4.5.7).

Для ограничения фиксированного типа с ограничением диапазона модельные числа включают нуль и все числа, кратные *дискрету, мантисса* которых может быть выражена точно В двоичными цифрами, а значение В выбрано как наименьшее целое число, для которого каждая граница заданного диапазона – либо модельное число, либо отличается не более чем на *дискрет* от модельного числа. Если ограничение фиксированного типа не включает ограничение диапазона (это допустимо только после обозначения типа в указании подтипа), модельные числа определены дельтой определения точности фиксированного типа и диапазоном подтипа, заданного обозначением типа.

Реализация должна иметь по крайней мере один анонимный предопределенный фиксированный тип. Базовый тип каждого такого фиксированного типа – это сам этот тип. Модельные числа каждого предопределенного фиксированного типа включают нуль и все числа, для которых *мантисса* (в канонической форме) имеет число двоичных цифр, вырабатываемое атрибутом MANTISSA, и для которых *дискрет* имеет значение, возвращаемое атрибутом SMALL.

Описание фиксированного типа в форме:

type T **is delta** Д **range** Л . . П;

по определению эквивалентно следующим описаниям:

type *фиксированный_тип* **is new**

предопределенный_фиксированный_тип;

subtype T **is** *фиксированный_тип* **delta** Д **range**

фиксированный_тип (Л) . . *фиксированный_тип* (П);

В этих описаниях *фиксированный_тип* – это анонимный тип, а предопределенный фиксированный тип неявно выбран реализацией так, чтобы его модельные числа включали определенные ограничением фиксированного типа модельные числа (т. е. с помощью Д, Л и П, а, возможно, и спецификатора длины, определяющего *дискрет*).

Описание фиксированного типа неправильно, если не существует предопределенного типа, удовлетворяющего этим требованиям. Хранимые числа фиксированного типа – это модельные числа его базового типа.

Предвыполнение описания фиксированного типа состоит из предвыполнения эквивалентных описаний типа и подтипа.

Если ограничение фиксированного типа следует за обозначением типа в указании подтипа, то обозначение типа должно задавать фиксированный тип или подтип. Ограничение фиксированного типа *совместимо* с обозначением типа, только если дельта из определения точности фиксированного типа не меньше дельты для типа или подтипа, заданного обозначением типа. Более того, если ограничение фиксированного типа включает ограничение диапазона, то ограничение фиксированного типа совместимо с обозначением типа, только если само ограничение совместимо с обозначением типа.

Предвыполнение такого указания подтипа включает предвыполнение ограничения диапазона, если оно есть, и создает фиксированный подтип, модельные числа которого определены соответствующим ограничением фиксированного типа, а также спецификатором длины, задающим дискрет, если он есть. Значение фиксированного типа принадлежит фиксированному подтипу тогда и только тогда, когда оно принадлежит диапазону, определенному подтипом.

Для всех фиксированных типов предопределены одни и те же арифметические операции (см. 4.5). Умножение и деление значений фиксированного типа дают результаты анонимного предопределенного фиксированного типа, который в данном стандарте называется *универсальный_фиксированный*; точность этого типа произвольна. Значения этого типа должны быть явно преобразованы в значения некоторого числового типа.

Примечание. Если С – подтип фиксированного типа или подтипа Т, то набор модельных чисел С – это подмножество модельных чисел Т. Если для Т был задан спецификатор длины, то Т и С имеют одно и то же значение *дискрета*. В противном случае, поскольку *дискрет* равен степени двух, то *дискрет* для С равен *дискрету* для Т, умноженному на неотрицательную степень двух.

Ограничение диапазона допустимо в указании фиксированного подтипа либо непосредственно за обозначением типа, либо как часть ограничения фиксированного типа. В обоих случаях границы диапазона должны принадлежать базовому типу обозначения типа (см. 3.5).

Примеры:

```
type ВОЛЬТ is delta 0.125 range 0.0. .255.0;
subtype ВОЛЬТ_ГРУБО is ВОЛЬТ delta 1.0;
- - диапазон как у ВОЛЬТ.
- - Правильная дробь, требующая полного машинного слова в
- - дополнительном коде, может быть описана как тип ДРОБЬ:
ДЕЛЬ: constant: = 1.0/2 ** (ДЛИНА_СЛОВА –1);
type ДРОБЬ is delta ДЕЛЬ range –1.0. .1.0 – ДЕЛЬ;
```

3.5.10. Операции над фиксированными типами

Базовые операции над фиксированным типом включают присваивание, проверку принадлежности, квалификацию, явное преобразование значений других числовых типов в значения этого фиксированного типа и неявное преобразование значений типа *универсальный_вещественный* в значение этого типа.

Кроме того, для каждого фиксированного типа или подтипа Т базовые операции включают перечисленные ниже атрибуты. В этом представлении атрибутов Т ссылаются на подтип (подтип Т) для любого свойства, зависящего от ограничений, наложенных на Т; другие свойства установлены в терминах базового типа Т.

Первая группа атрибутов вырабатывает характеристики подтипа Т. К этой группе относятся атрибуты BASE (см. 3.3.3), FIRST и LAST (см. 3.5), атрибут представления SIZE (см. 13.7.2), а также атрибуты:

T'DELTA Вырабатывает значение дельты, заданной в определении точности фиксированного типа для подтипа Т. Значение этого атрибута имеет тип *универсальный_вещественный.*

T'MANTISSA Вырабатывает число двоичных цифр в мантиссе модельных чисел подтипа Т. (Этот атрибут вырабатывает число B, см. разд. 3.5.9). Значение этого атрибута имеет тип *универсальный_целый.*

T'SMALL Вырабатывает наименьшее положительное (ненулевое) модельное число подтипа Т. Значение этого атрибута имеет тип *универсальный_вещественный.*

T'LARGE Вырабатывает наибольшее положительное модельное число подтипа Т. Значение этого атрибута имеет тип *универсальный_вещественный.*

T'FORE Вырабатывает минимальное число символов, необходимых для десятичного представления целой части любого значения подтипа Т в предположении, что это представление не включает порядок, но включает один символ, который является либо знаком минус, либо пробелом. (Это минимальное число учитывает незначащие нули и подчеркивания и по меньшей мере равно двум.) Значение этого атрибута имеет *универсальный_целый* тип.

T'AFT Вырабатывает число десятичных цифр после точки, необходимых для обеспечения точности подтипа Т, если только DELTA подтипа Т больше 0.1, а для 0.1 атрибут вырабатывает значение единица. (T'AFT – это самое малое положительное целое К, для которого (10 ** К)* T'DELTA больше или равно единице.) Значение этого атрибута имеет тип *универсальный_целый.*

Вторая группа включает следующие атрибуты, они вырабатывают характеристики хранимых чисел:

T'SAFE_SMALL Вырабатывает наименьшее положительное (ненулевое) хранимое число базового типа Т. Значение этого атрибута имеет тип *универсальный_вещественный.*

T'SAFE_LARGE Вырабатывает наибольшее положительное хранимое число базового типа Т. Значение этого атрибута имеет тип *универсальный_вещественный.*

Кроме того, для объекта А фиксированного типа определены атрибуты A'SIZE и A'ADDRESS (см. 13.7.2). Для каждого фиксированного типа или подтипа Т существуют машинно-зависимые атрибуты T'MACHINE_ROUNDS и T'MACHINE_OVERFLOWS (см. 13.7.3).

Кроме базовых, в состав операций над фиксированным типом входят операции отношения и следующие предопределенные арифметические операции: унарные и бинарные аддитивные операции – и +, мультипликативные операции * и / и операция **abs**.

Операции над подтипом – это соответствующие операции типа, кроме следующих: присваивания, проверки принадлежности, квалификации, явного преобразования и атрибутов первой группы; результат этих операций определен в терминах подтипа.

Примечание. Значение атрибута T'FORE зависит только от диапазона подтипа Т. Значение атрибута T'AFT зависит только от значения T'DELTA. Между атрибутами фиксированного типа существуют следующие соотношения:

T'LARGE = (2 ** T'MANTISSA – 1) * T'SMALL
T'SAFE_LARGE = T'BASE'LARGE
T'SAFE_SMALL = T'BASE'SMALL

3.6. **Индексируемые типы**

Объект индексируемого типа (массив) – это составной объект, содержащий компоненты одного и того же подтипа. В имени компонента массива используется одно или несколько индексных значений, принадлежащих заданным дискретным типам. Значение массива – это составное значение, состоящее из значений его компонентов.

определение_индексируемого_типа : : =
 определение_неограниченного_индексируемого_типа
 |определение_ограниченного_индексируемого_типа
определение_неограниченного_индексируемого_типа : : =
 array (определение_подтипа_индекса
 {, определение_подтипа_индекса}) **of**
 указание_подтипа_*компонента*
определение_ограниченного_индексируемого_типа : : =
 array ограничение_индекса **of**
 указание_подтипа_компонента
определение_подтипа_индекса : : =
 обозначение_типа **range** < >
ограничение_индекса : : =
 (дискретный_диапазон {, дискретный_диапазон})
дискретный_диапазон : : =
 указание_*дискретного*_подтипа | диапазон

Массив характеризуется числом индексов (*размерность* массива), типом и позицией каждого индекса, верхней и нижней границами каждого индекса, а также типом и возможным ограничением компонентов. Порядок индексов существенен.

Для каждого возможного значения индекса одномерный массив имеет отдельный компонент. Многомерный массив имеет отдельный компонент для каждой возможной последовательности значений индексов, которая может быть образована фиксацией значений для каждой позиции индекса (в данном порядке). Возможными значениями индекса являются все значения между нижней и верхней границами включительно; этот диапазон значений называется *диапазоном индекса.*

Определение неограниченного индексируемого типа определяет индексируемый тип. Для каждого объекта индексируемого типа число индексов, тип и позиция каждого индекса, а также подтип компонентов, будут такими, как в определении типа; значения нижней и верхней границ для каждого индекса принадлежат соответствующему подтипу индекса, кроме пустых массивов, как пояснено в разд. 3.6.1. *Подтипом индекса* для данной позиции индекса по определению является подтип, указанный обозначением типа соответствующего определения подтипа индекса. Составной ограничитель < > (бокс) в определении подтипа индекса помещается для неопреде-

ленного диапазона (различные объекты данного типа не обязательно имеют одни и те же границы). Предвыполнение определения неограниченного индексируемого типа создает индексируемый тип: оно включает предвыполнение указания подтипа компонентов.

Определение ограниченного индексируемого типа определяет индексируемый тип и подтип этого типа:

• Индексируемый тип – это неявно описанный анонимный тип; этот тип определен (неявно) определением неограниченного индексируемого типа, в котором указание подтипа компонентов берется из определения ограниченного индексируемого типа и в котором обозначение типа каждого определения подтипа по каждому индексу определяется соответствующим дискретным диапазоном.

• Индексируемый подтип – это подтип, полученный наложением ограничения индекса на индексируемый тип.

Если определение ограниченного индексируемого типа дано в описании типа, то простое имя, введенное этим описанием, обозначает индексируемый подтип.

Предвыполнение определения ограниченного индексируемого типа создает соответствующий индексируемый тип и индексируемый подтип. При этом предвыполняются ограничение индекса и указание подтипа компонентов. Вычисление каждого дискретного диапазона ограничения индекса и предвыполнение указания подтипа компонентов осуществляется в порядке, не определяемом языком.

Примеры описаний типа с определениями неограниченного индексируемого типа:

```
type ВЕКТОР is array (INTEGER range < >) of ВЕЩЕСТВ;
type МАТРИЦА is array (INTEGER range < >, INTEGER range < >) of ВЕЩЕСТВ;
type ВЕКТОР_БИТ is array (INTEGER range < >) of BOOLEAN;
type РИМСКИЙ is array (POSITIVE range < >) of РИМСКАЯ_ЦИФРА;
```

Примеры описаний типа с определениями ограниченного индексируемого типа:

```
type ТАБЛИЦА is array (1 . . 10) of INTEGER;
type РАСПИСАНИЕ is array (ДЕНЬ) of BOOLEAN;
type СТРОЧКА is array (1. . МАКС_РАЗМЕР_СТРОЧКИ) of CHARACTER;
```

Примеры описаний объектов с определениями ограниченного индексируемого типа:

```
РЕШЕТКА: array (1. .80, 1. .100) of BOOLEAN;
СМЕСЬ: array (ЦВЕТ range КРАСНЫЙ. .ЗЕЛЕНЫЙ) of BOOLEAN;
СТРАНИЦА: array (1. .50) of СТРОЧКА; - - массив массивов.
```

Примечание. Для одномерного массива приведенное правило означает, что описание с определением ограниченного индексируемого типа, например:

```
type T is array (POSITIVE range МИН. .МАКС)
                 of КОМПОНЕНТОВ;
```

эквивалентно (при отсутствии некорректной зависимости от порядка) последовательности описаний:

```
subtype подтип_индекс is POSITIVE range МИН . . МАКС;
```

type *индексируемый_тип* is array (*подтип_индекса* range < >)
of КОМПОНЕНТОВ;
subtype T is *индексируемый_тип (подтип_индекса);*

где *подтип_индекса* и *индексируемый_тип* оба анонимны. Следовательно, T – имя подтипа, и все объекты, описанные с этим обозначением типа, – массивы, имеющие одни и те же границы. Аналогичные преобразования применяются к многомерным массивам.

Подобное преобразование применяется к объекту, описание которого включает определение ограниченного индексируемого типа. Следствием этого является то, что нет двух таких объектов одного и того же типа.

3.6.1. Ограничения индекса и дискретные диапазоны

Ограничение индекса определяет диапазон возможных значений каждого индекса индексируемого типа и, таким образом, соответствующие границы массива.

Для дискретного диапазона, использованного в определении ограниченного индексируемого типа и определенного диапазоном, неявное преобразование к предопределенному типу INTEGER производится в том случае, если каждая граница – это либо числовой литерал, либо именованное число, либо атрибут, а тип обеих границ (до неявного преобразования) является *универсальным_целым.* В остальных случаях обе границы должны быть одного и того же дискретного типа, отличного от типа *универсальный_целый;* этот тип должен определяться независимо от контекста, но с учетом того, что тип должен быть дискретным и что обе границы должны иметь один и тот же тип. Эти правила применимы также к дискретному диапазону, используемому в правиле итерации (см. 5.5) или в описании семейства входов (см. 9.5).

Если ограничение индекса следует за обозначением типа в указании подтипа, то тип или подтип, указанный обозначением типа, не должен содержать ограничение индекса. Обозначение типа должно указывать либо неограниченный индексируемый тип, либо ссылочный тип, указываемый тип которого – такой же индексируемый тип. В любом случае, ограничение индекса должно задавать дискретный диапазон для каждого индекса индексируемого типа, и тип каждого дискретного диапазона должен быть тем же самым, что и у соответствующего индекса.

Ограничение индекса *совместимо* с типом, указанным в обозначении типа, если и только если ограничение, определенное каждым дискретным диапазоном, совместимо с соответствующим подтипом индекса. Если какой-нибудь из дискретных диапазонов определяет пустой диапазон, то ограниченный таким образом массив является *пустым массивом,* не имеющим компонентов. Значение массива *удовлетворяет* ограничению индекса, если в каждой позиции индекса значение массива и ограничение индекса имеют одни и те же границы индекса. (Заметим, однако, что присваивание и некоторые другие операции над массивами включают неявное преобразование подтипа.)

Границы каждого массива определены следующим образом:

• Для заданной описанием объекта переменной указание подтипа соответствующего описания объекта должно определять подтип ограниченного индексируемого типа (и, таким образом, границы). То же самое требуется от указания подтипа описания компонента, если тип компонента записи – индексируемый тип, а также от указания подтипа компонента определения индексируемого типа, если тип компонентов массива является сам индексируемым типом.

• Для заданной описанием объекта константы индексируемого типа границы определены начальным значением, если подтип константы неограничен; иначе они определены подтипом (в последнем случае начальное значение – это результат неявного преобразования подтипа). То же правило применимо к формальному параметру настройки вида in.

• Для указанного ссылочным значением массива границы должны быть определены генератором, создающим массив. (Созданный объект ограничен соответствующими значениями границ).

• Для формального параметра подпрограммы или входа границы получены от соответствующего фактического параметра. (Формальный параметр ограничен соответствующими значениями границ).

• Для описания переименования и для формального параметра настройки вида in out границы берутся у переименованного объекта или у соответствующего фактического параметра настройки.

Порядок вычисления дискретных диапазонов при предвыполнении ограничения индекса в языке не определен.

Примеры описаний массивов, включающих ограничение индексов:

```
ПУЛЬТ: МАТРИЦА (1. .8, 1. .8); - - см. 3.6
ПРЯМОУГОЛЬНИК: МАТРИЦА (1. .20, 1. .30);
ОБРАТНАЯ: МАТРИЦА (1. .К, 1. .К); - - К не обязательно статическое
ФИЛЬТР: ВЕКТОР_БИТ (0. .31);
```

Пример описания массива с ограниченным индексируемым подтипом:

```
МОЕ_РАСПИСАНИЕ: РАСПИСАНИЕ; - - все массивы типа
    - - РАСПИСАНИЕ имеют одни и те же границы.
```

Пример именуемого типа с компонентом индексируемого типа:

```
type ПЕРЕМ_СТРОЧКА (ДЛИНА: INTEGER) is
  record
     ОБРАЗ: STRING (№. .ДЛИНА);
  end record;
ПУСТАЯ_СТРОЧКА: ПЕРЕМ_СТРОЧКА (0);
- - ПУСТАЯ_СТРОЧКА. ОБРАЗ – пустой массив
```

Примечание. Предвыполнение указания подтипа, состоящего из обозначения типа, за которым следует ограничение индекса, контролирует совместимость ограничения индекса с обозначением типа (см. 3.3.2).

Все компоненты массива имеют один и тот же подтип. В частности, для массивов, компоненты которых являются одномерными массивами, это означает, что все компоненты имеют одни и те же границы, и, следовательно, одинаковую длину.

3.6.2. Операции над индексируемыми типами

Базовые операции над индексируемым типом включают присваивание, агрегаты (если индексируемый тип не является лимитируемым типом), про-

верку принадлежности, индексируемые компоненты, квалификацию и явное преобразование; для одномерных массивов базовые операции включают также отрезки и строковые литералы, если тип компонентов – символьный тип.

Если А – объект, являющийся массивом, значение массива или подтип ограниченного индексируемого типа, то базовые операции включают также атрибуты, которые перечислены ниже. Эти атрибуты недопустимы для неограниченного индексируемого типа. Аргумент К, использованный в обозначении атрибута для К-го измерения массива, должен быть положительным (ненулевым) статическим выражением типа *универсальный_целый* и не больше размерности массива.

A'FIRST Вырабатывает значение нижней границы диапазона первого индекса. Значение этого атрибута имеет тот же тип, что и эта нижняя граница.

A'FIRST(K) Вырабатывает значение нижней границы диапазона К-го индекса. Значение этого атрибута имеет тот же тип, что и эта нижняя граница.

A'LAST Вырабатывает значение верхней границы диапазона первого индекса. Значение этого атрибута имеет тот же тип, что и эта верхняя граница.

A'LAST(K) Вырабатывает значение верхней границы диапазона К-го индекса. Значение этого атрибута имеет тот же тип, что и эта верхняя граница.

A'RANGE Вырабатывает диапазон первого индекса, т. е. диапазон A'FIRST . . A'LAST.

A'RANGE(K) Вырабатывает диапазон К-го индекса, т. е. диапазон A'FIRST(K) . .A'LAST(K).

A'LENGTH Вырабатывает количество значений в диапазоне первого индекса (нуль для пустого диапазона). Значение этого атрибута имеет тип *универсальный_целый.*

A'LENGTH(K) Вырабатывает количество значений в диапазоне К-го индекса (нуль для пустого диапазона). Значение этого атрибута имеет тип *универсальный_целый.*

Кроме того, для индексируемого типа или подтипа Т определены атрибуты T'BASE и T'SIZE (см. 3.3.3), а для массива А определены атрибуты A'SIZE и A'ADDRESS (см. 13.7.2).

Кроме базовых, операции над индексируемым типом включают предопределенные операции сравнения на равенство или неравенство, если индексируемый тип не является лимитируемым типом. Для одномерных массивов к этим операциям относится катенация, если индексируемый тип не является лимитируемым типом; если тип компонента – дискретный тип, то операции включают также все предопределенные операции отношения; если тип компонента – логический тип, то операции также включают унарную логическую операцию отрицания и логические операции.

Примеры (использующие массивы, описанные в примерах разд. 3.6.1):

```
--ФИЛЬТР'FIRST = 0
--ФИЛЬТР'LAST = 31
--ФИЛЬТР'LENGTH = 32
--ПРЯМОУГОЛЬНИК'LAST(1) = 20
--ПРЯМОУГОЛЬНИК'LAST(2) = 30
```

Примечание. Атрибуты A'FIRST и A'FIRST(1) вырабатывают одно и то же значение. То же можно сказать об атрибутах A'LAST, A'RANGE и A'LENGTH. Для приведенных атрибутов удовлетворяются следующие соотношения (исключая пустой массив), если тип индекса – целый тип:

A'LENGTH = A'LAST – A'FIRST +1

A'LENGTH(K) = A'LAST(K) – A'FIRST(K) +1

Индексируемый тип – лимитируемый, если тип его компонентов – лимитируемый (см. 7.4.4).

3.6.3. Строковый тип

Значения предопределенного типа STRING – это одномерные массивы компонентов предопределенного типа CHARACTER, индексируемые значениями предопределенного подтипа POSITIVE:

```
subtype POSITIVE is INTEGER range 1. .INTEGER'LAST;
type STRING is array (POSITIVE range < >) of CHARACTER;
```

Примеры:

```
ЗВЕЗДОЧКИ: STRING (1. .120) : = (1. .120 = > '*');
ВОПРОС: constant STRING : = "СКОЛЬКО СИМВОЛОВ?";
-- ВОПРОС'FIRST = 1, ВОПРОС'LAST = 17 (число символов)
ДВОЙНОЙ_ВОПРОС: constant STRING: = ВОПРОС & ВОПРОС;
ДЕВЯНОСТО_ШЕСТЬ: constant РИМСКИЙ: = "XCVI"; -- см. 3.6
```

Примечание. Строковые литералы (см. 2.6 и 4.2) – базовые операции над типом STRING и любым другим одномерным индексируемым типом, тип компонентов которого – символьный тип. Операция катенации – предопределенная операция над типом STRING и для одномерных индексируемых типов; она представляется знаком &. Операции отношения <, <=, > и >= определены для значений этих типов в соответствии с лексикографическим порядком (см. 4.5.2).

3.7. **Именуемые типы**

Объект именуемого типа (запись) – это составной объект, состоящий из именованных компонентов. Значение записи – составное значение, состоящее из значений своих компонентов.

```
определение_именуемого_типа : : =
  record
     список_компонентов
  end record
список_компонентов : : =
  описание_компонента {описание_компонента}
  | {описание_компонента} раздел_вариантов | null;
описание_компонента : : =
  список_идентификаторов : определение_подтипа_компонента
  [:= выражение] ;
определение_подтипа_компонента : : = указание_подтипа
```

Каждое описание компонента задает компонент именуемого типа. Кроме этих компонентов, в компоненты именуемого типа включаются любые компоненты, описанные спецификациями дискриминантов в описании именуемого типа. Идентификаторы всех компонентов именуемого типа должны быть различными. Использование имени, обозначающего компонент записи, отличный от дискриминанта, недопустимо в том же определении именуемого типа, содержащем описание этого компонента.

Описание компонента с несколькими идентификаторами эквивалентно последовательности единичных описаний компонентов, как пояснено в разд. 3.2. Каждое единичное описание компонента описывает компонент записи, подтип которого задан определением подтипа компонента.

Если описание компонента включает составной ограничитель присваивания, за которым следует выражение, то это выражение является выражением по умолчанию для компонента записи; выражение по умолчанию должно иметь тип компонента. Выражения по умолчанию недопустимы для компонентов лимитируемого типа.

Если именуемый тип не имеет раздела дискриминантов, то одни и те же компоненты присутствуют во всех значениях этого типа. Если список компонентов именуемого типа определен зарезервированным словом **null** и раздел дискриминантов отсутствует, то именуемый тип не имеет компонентов, и все записи этого типа – *пустые* записи.

Предвыполнение определения именуемого типа создает именуемый тип; оно состоит из предвыполнения соответствующих (единичных) описаний компонентов в порядке их следования, включая описания компонентов в разделе вариантов. Предвыполнение описания компонента состоит из предвыполнения определения подтипа компонента.

Для предвыполнения определения подтипа компонента в случае, когда ограничение не зависит от дискриминанта (см. 3.7.1), предвыполняется указание подтипа. Если, с другой стороны, ограничение зависит от дискриминанта, то предвыполнение состоит из вычисления каждого входящего в ограничение выражения, которое не является дискриминантом.

Примеры описаний именуемого типа:

```
type ДАТА is
  record
     ДЕНЬ: INTEGER range 1. .31;
     МЕСЯЦ: ИМЯ_МЕСЯЦА;
     ГОД: INTEGER range 0. .4000;
  end record;
type КОМПЛЕКСНЫЙ is
  record
     ВЕЩ: ВЕЩЕСТВ: = 0.0;
     МНИМ: ВЕЩЕСТВ: = 0.0;
  end record;
```

Примеры переменных именуемого типа:

```
ЗАВТРА, ВЧЕРА: ДАТА;
A, B, C: КОМПЛЕКСНЫЙ;
- - компоненты A, B и C неявно инициализированы нулем.
```

Примечание. Выражения по умолчанию для компонента при отсутствии явной инициализации записи неявно вычисляются при предвыполнении описания записи (см. 3.2.1). Если описание компонента имеет несколько идентификаторов, то выражение вычисляется один раз для каждого такого компонента записи (поскольку это описание эквивалентно последовательности единичных описаний компонентов).

В отличие от компонентов массива компоненты записи не обязательно имеют один и тот же тип.

3.7.1. Дискриминанты

Раздел дискриминантов специфицирует дискриминанты типа. Дискриминант записи – это компонент записи. Тип дискриминанта должен быть дискретным.

раздел_дискриминантов : : =
(спецификация_дискриминанта
{; спецификация_дискриминанта})
спецификация_дискриминанта : : =
список_идентификаторов : обозначение_типа
[: = выражение]

Раздел дискриминантов допустим только в описании именуемого типа, в описании личного типа или в неполном описании типа (соответствующее полное описание должно тогда описывать именуемый тип), а также в описании параметра настройки для формального личного типа.

Спецификация дискриминанта с несколькими идентификаторами эквивалентна последовательности единичных спецификаций дискриминантов, как пояснено в разд. 3.2. Каждая единичная спецификация дискриминанта описывает дискриминант. Если спецификация дискриминанта включает составной ограничитель присваивания, за которым следует выражение, то это – выражение по умолчанию для дискриминанта; выражения по умолчанию должны быть заданы либо для всех, либо не заданы ни для одного из дискриминантов раздела дискриминантов.

Использование имени дискриминанта недопустимо в выражениях по умолчанию раздела дискриминантов, если спецификация дискриминанта сама дана в этом разделе дискриминантов.

В определении именуемого типа допустимы только следующие случаи использования имени дискриминанта: в выражениях по умолчанию для компонентов записи, в разделе вариантов в качестве имени дискриминанта, а также в определении подтипа компонента – либо как граница в ограничении индекса, либо для задания значения дискриминанта в ограничении дискриминанта. Использованное в этих определениях подтипа компонента имя дискриминанта должно встречаться само по себе, а не как часть большего выражения. Такие определения подтипа компонентов и такие ограничения называются *зависимыми от дискриминанта.*

Компонент называется *зависимым от дискриминанта,* если он – либо компонент записи, описанный в разделе вариантов, либо компонент записи, чье определение подтипа компонента зависит от дикриминанта, либо, наконец, один из подкомпонентов компонента, который сам зависит от дискриминанта.

Каждое значение записи включает значение каждого дискриминанта, заданного для именуемого типа; оно также включает значение для каждого компонента записи, который не зависит от дискриминанта. Значения дискриминантов определяют, значения каких компонентов, зависящих от дискриминантов, присутствуют в значении записи.

Непосредственное присваивание значения дискриминанту объекта недопустимо; более того, дискриминант недопустим как фактический параметр вида **in out** или **out**, либо как фактический параметр настройки вида **in out**. Единственным допустимым способом изменения значения дискриминанта переменной является присваивание (полного) значения самой переменной. Аналогично, присваивание самой переменной является единственным допустимым путем изменения ограничения одного из ее компонентов, если определение подтипа компонента зависит от дискриминанта переменной.

Предвыполнение раздела дискриминантов не имеет другого эффекта.

Примеры:

```
type БУФЕР (РАЗМЕР: РАЗМЕР_БУФЕРА: = 100) is - - см. 3.5.4
  record
     ПОЗ: РАЗМЕР_БУФЕРА: = 0;
     ЗНАЧЕНИЕ: STRING (1. .РАЗМЕР);
  end record;
type КВАДРАТ (СТОРОНА: INTEGER) is
  record
     МАТР: МАТРИЦА (1. .СТОРОНА, 1. .СТОРОНА); - - см. 3.6
  end record;
type ДВОЙНОЙ_КВАДРАТ (ЧИСЛО: INTEGER) is
  record
     ЛЕВЫЙ: КВАДРАТ (ЧИСЛО);
     ПРАВЫЙ: КВАДРАТ (ЧИСЛО);
  end record;
type ЭЛЕМЕНТ (ЧИСЛО: POSITIVE) is
  record
     СОДЕРЖАНИЕ: INTEGER;
     - - компонент не зависит от дискриминанта
  end record;
```

3.7.2. Ограничения дискриминантов

Ограничение дискриминанта допустимо только в указании подтипа за обозначением типа. Это обозначение типа должно указывать либо тип с дискриминантами, либо ссылочный тип, обозначающий тип с дискриминантом. Ограничение дискриминанта задает значения этих дискриминантов.

ограничение_дискриминанта : : =
  (сопоставление_дискриминанта
  {, сопоставление_дискриминанта})
сопоставление_дискриминанта : : =
  [простое_имя_*дискриминанта*
  {|простое_имя_*дискриминанта*} = >] выражение

Каждое сопоставление дискриминанта связывает выражение с одним или несколькими дискриминантами. Сопоставление дискриминанта называется *именованным*, если дискриминанты заданы явно своими именами; иначе оно называется *позиционным*. Для позиционного сопоставления (единственный) дискриминант неявно задан позицией в текстуальном порядке. Именованные сопоставления могут быть даны в любом порядке, но если в одном и том же ограничении дискриминанта использованы позиционные и именованные сопоставления, то позиционные сопоставления должны

стоять первыми в их обычной позиции. Следовательно, если однажды было использовано именованное сопоставление, то в остальной части ограничения дискриминанта должны употребляться только именованные сопоставления.

Для именованного сопоставления дискриминанта имена дискриминантов должны обозначать дискриминанты типа, для которого дано ограничение дискриминанта. Сопоставление дискриминанта с более чем одним именем дискриминанта допустимо только, если все поименованные дискриминанты имеют один и тот же тип. Более того, для каждого сопоставления дискриминанта (именованного или позиционного) выражение и сопоставленные дискриминанты должны иметь один и тот же тип. Ограничение дискриминанта должно задавать точно одно значение для каждого дискриминанта типа.

Ограничение дискриминанта совместимо с типом, указанным обозначением типа, если и только если каждое значение дискриминанта принадлежит подтипу соответствующего дискриминанта. Кроме того, для каждого подкомпонента, описание подтипа которого зависит от дискриминанта, значение дискриминанта в этой позиции описания подтипа компонента заменяется значением дискриминанта и производится проверка совместимости результирующего указания подтипа.

Составное значение удовлетворяет ограничению дискриминанта, если и только если каждый дискриминант составного значения имеет налагаемое ограничением дискриминанта значение.

Начальные значения дискриминантов объекта, имеющего тип с дискриминантами, определяются следующим образом:

· Для переменной, заданной описанием объекта, указание подтипа должно быть ограничено ограничением дискриминанта, если выражения по умолчанию для дискриминантов отсутствуют; значения дискриминантов определены либо ограничением, либо, при отсутствии его, выражением по умолчанию. То же самое требование существует для указания подтипа в описании компонента, если тип компонента записи имеет дискриминанты, а также для указания подтипа компонента индексируемого типа, если тип компонента массива – тип с дискриминантами.

· Для константы, заданной описанием объекта, значения дискриминантов берутся из начального значения, если подтип константы неограничен; иначе они определяются из этого подтипа (в последнем случае возбуждается исключение, если начальное значение не принадлежит этому подтипу). То же правило применяется к параметру настройки вида in.

· Для объекта, указанного ссылочным значением, значения дискриминантов должны быть определены генератором, создающим объект. (Созданный объект ограничен соответствующими значениями дискриминантов).

· Для формального параметра подпрограммы или входа дискриминанты формального параметра инициализированы значениями дискриминантов соответствующего фактического параметра. (Формальный параметр ограничен, если соответствующий фактический параметр ограничен, и ограничен в любом случае, если его вид in или если подтип формального параметра ограничен).

• Для описания переименования и для формального параметра настройки вида **in out** дискриминанты – это дискриминанты переименованного объекта или соответствующего фактического параметра настройки.

Порядок вычислений выражений, данных в сопоставлениях дискриминантов, при предвыполнении ограничения дискриминанта в языке не определен; выражение именованного сопоставления вычисляется по одному разу для каждого именованного дискриминанта.

Примеры (использующие типы, описанные в предыдущем разделе):

```
БОЛЬШОЙ: БУФЕР (200); -- ограничен, всегда 200 символов
                      -- (явное значение дискриминанта)
СООБЩЕНИЕ: БУФЕР;     -- неограничен, вначале 100 символов
                      -- (значение дискриминанта по умолчанию)
БАЗИС: КВАДРАТ (5);   -- ограничен, всегда 5 на 5
НЕПРАВИЛЬНЫЙ: КВАДРАТ; -- неправильно, КВАДРАТ должен быть ограничен
```

Примечание. Приведенные правила и правила, определяющие предвыполнение описания объекта (см. 3.2), гарантируют, что дискриминанты всегда имеют значения. В частности, если ограничение дискриминанта входит в описание объекта, то каждый дискриминант инициализован значением, определяемым ограничением. Аналогично, если подтип компонента имеет ограничение дискриминанта, то дискриминанты этого компонента соответственно инициализованы.

3.7.3. Разделы вариантов

Именуемый тип с разделом вариантов задает альтернативные списки компонентов. Каждый вариант определяет компоненты для соответствующего значения или значений дискриминанта.

```
раздел_вариантов : : =
  case простое_имя_дискриминанта is
     вариант
     {вариант}
  end case;
вариант : : =
  when выбор {|выбор} =>
     список_компонентов
выбор : : = простое_выражение
  | дискретный_диапазон | others
  | простое_имя_компонента
```

Каждый вариант начинается со списка выборов, которые должны быть того же типа, что и дискриминант раздела вариантов. Тип дискриминанта раздела вариантов не должен быть формальным типом настройки. Если подтип дискриминанта статический, то каждое значение этого подтипа должно быть представлено в наборе выборов раздела вариантов один и только один раз, и никакие другие значения не допустимы. В противном случае, каждое значение (базового) типа дискриминанта должно быть представлено в наборе вариантов один и только один раз.

Простые выражения и дискретные диапазоны, данные как выборы в разделе вариантов, должны быть статическими. Определенный дискретным диапазоном выбор задает все значения соответствующего диапазона (и ни

одного, если диапазон пустой). Выбор **others** допустим только для последнего варианта и только как его единственный выбор; он задает все остальные значения (возможно и ни одного), не упомянутые в выборах предыдущих вариантов. Простое имя компонента недопустимо в качестве выбора варианта (хотя оно присутствует в синтаксисе выбора).

Значение записи содержит значения компонентов данного варианта тогда и только тогда, когда значение дискриминанта равно одному из значений, заданных выборами этого варианта. Это правило применимо, в свою очередь, к любому вложенному варианту, который сам включен в список компонентов данного варианта. Если список компонентов варианта задан как **null**, то вариант не имеет компонентов.

Примеры именуемого типа с разделом вариантов:

```
type УСТРОЙСТВО is (ПЕЧАТАЮЩЕЕ_УСТРОЙСТВО, ДИСК, БАРАБАН);
type СОСТОЯНИЕ is (ОТКРЫТ, ЗАКРЫТ);
type ПЕРИФЕРИЙНЫЙ (МОДУЛЬ: УСТРОЙСТВО: = ДИСК) is
  record
     СТАТУС: СОСТОЯНИЕ;
     case МОДУЛЬ is
       when ПЕЧАТАЮЩЕЕ_УСТРОЙСТВО =>
         СЧЕТ_СТРОЧЕК: INTEGER range 1. .РАЗМЕР_СТР;
       when others = >
         ЦИЛИНДР: ИНДЕКС_ЦИЛИНДРА;
         ТРАКТ: НОМЕР_ТРАКТА;
     end case;
  end record;
```

Примеры подтипов записей:

```
subtype УСТРОЙСТВО_БАРАБАНА is ПЕРИФЕРИЙНЫЙ (БАРАБАН);
subtype УСТРОЙСТВО_ДИСКА is ПЕРИФЕРИЙНЫЙ (ДИСК);
```

Примеры ограниченных переменных именуемого типа:

```
ПИШУЩЕЕ_УСТРОЙСТВО: ПЕРИФЕРИЙНЫЙ (МОДУЛЬ =>
                                  ПЕЧАТАЮЩЕЕ_УСТРОЙСТВО);
АРХИВ: УСТРОЙСТВО_ДИСКА;
```

Примечание. Выборы с дискретными значениями используются также в операторах и в агрегатах массива. Выборы с простыми именами компонентов используются только в агрегатах записей.

3.7.4. Операции над именуемыми типами

Базовые операции над именуемым типом включают присваивание и агрегаты (если тип – не лимитируемый тип), проверку принадлежности, именование компонентов записи, квалификацию и преобразование типа (для производных типов).

Для любого объекта А типа с дискриминантами базовые операции включают также следующий атрибут.

A'CONSTRAINED Вырабатывает значение TRUE, если ограничение дискриминанта наложено на объект А или если объект – константа (включая формальный параметр или формальный параметр настройки вида **in**); вырабатывает значение FALSE в противном случае. Если А – формальный параметр настройки вида **in out** или если А – формальный параметр вида **in**

out или **out**, и данное в соответствующей спецификации параметра обозначение типа обозначает неограниченный тип с дискриминантами, то значение этого атрибута получается из значения атрибута соответствующего фактического параметра. Значение этого атрибута имеет предопределенный тип BOOLEAN.

Кроме того, атрибуты T'BASE и T'SIZE определены для именуемого типа или подтипа T (см. 3.3.3); атрибуты A'SIZE и A'ADDRESS определены для записи A (см. 13.7.2).

Кроме базовых, операции над именуемым типом включают предопределенное сравнение на равенство и неравенство, если тип не является лимитируемым.

Примечание. Именуемый тип – лимитируемый, если тип хотя бы одного из его компонентов – *лимитируемый* (см. 7.4.4).

3.8. **Ссылочные типы**

Объявленный описанием объект создается предвыполнением этого описания и обозначается простым именем или некоторой другой формой имени. В противоположность этому существуют объекты, создаваемые вычислением *генераторов* (см. 4.8) и не имеющие простого имени. Доступ к такому объекту осуществляется посредством возвращаемого генератором *ссылочного значения*; говорят, что ссылочное значение *указывает* объект.

определение_ссылочного_типа : : = **access** указание_подтипа

Для каждого ссылочного типа существует литерал **null**, имеющий пустое ссылочное значение. вообще не указывающее объект. Пустое значение ссылочного типа – начальное значение этого типа по умолчанию. Другие значения ссылочного типа получаются вычислением специальной операции над типом, называемой генератором. Каждое такое ссылочное значение указывает объект подтипа, обозначенного указанием подтипа определения ссылочного типа; этот подтип называется *указываемым подтипом*; базовый тип этого подтипа называется *указываемым типом.* Указанные значением ссылочного типа объекты образуют *набор,* неявно связанный с этим типом.

Предвыполнение определения ссылочного типа состоит из предвыполнения указания подтипа и создания ссылочного типа.

Если ссылочный объект – константа, то ссылочное значение не может быть изменено и всегда указывает один и тот же объект. С другой стороны, значение указываемого объекта не обязательно остается одним и тем же (присваивание указываемому объекту допустимо, если указываемый тип не лимитируемый).

Единственные формы ограничения, которые допустимы после имени ссылочного типа в указании подтипа, – это ограничения индексов и ограничения дискриминантов (см. разд. 3.6.1 и 3.7.2 для правил, применимых к этим указаниям подтипа). Ссылочное значение *принадлежит* соответствующему подтипу ссылочного типа, если либо ссылочное значение – пустое значение, либо значение указываемого объекта удовлетворяет ограничению.

Примеры:

type ОБРАМЛЕНИЕ **is access** МАТРИЦА; - - см. 3.6
type ИМЯ_БУФЕРА **is access** БУФЕР; - - см. 3.7.1

Примечание. Ссылочное значение, передаваемое генератором, может быть присвоено нескольким ссылочным объектам. Следовательно, объект, созданный генератором, может быть указан более чем одной переменной или константой ссылочного типа. Ссылочное значение может указывать только объект, созданный генератором, в частности, оно не может указывать объект, объявленный описанием объекта.

Если тип объектов, указанных ссылочными значениями, – индексируемый тип или тип с дискриминантами, то эти объекты ограничены либо границами массива, либо значениями дискриминантов, заданными неявно или явно соответствующими генераторами (см. 4.8).

Ссылочные значения в некоторых других языках называются *указателями* или *ссылками.*

3.8.1. Неполные описания типов

Никаких конкретных ограничений на тип, указываемый ссылочным типом, не существует. В частности, тип компонента указываемого типа может быть другим ссылочным типом или даже тем же самым ссылочным типом. Это позволяет вводить взаимозависимые и рекурсивные ссылочные типы. Их описания требуют предварительного неполного описания типа (или описания личного типа) для одного или нескольких типов.

неполное_описание_типа : : =
type идентификатор [раздел_дискриминантов] ;

Для каждого неполного описания типа должно быть соответствующее описание типа с тем же идентификатором. Соответствующее описание должно быть либо полным описанием, либо описанием задачного типа. В оставшейся части главы пояснения даны в терминах полных описаний типа; те же правила применяются к описаниям задачного типа. Если неполное описание типа встречается непосредственно в разделе описаний или видимом разделе спецификации пакета, то полное описание типа должно встретиться позже непосредственно в этом разделе описаний или видимом разделе. Если неполное описание типа встречается непосредственно в личном разделе пакета, то полное описание типа должно быть позже непосредственно в этом личном разделе или же в разделе описаний соответствующего тела пакета.

Раздел дискриминантов должен быть дан в полном описании типа тогда и только тогда, когда он дан в неполном описании типа; если разделы дискриминантов даны, то они должны быть согласованы (см. 6.3.1 для правил согласования). До конца полного описания типа использование имени, объявленного неполным описанием типа, допустимо только как обозначение типа в указании подтипа определения ссылочного типа; единственной формой ограничения, допустимой в указании подтипа, являются ограничения дискриминанта.

Предвыполнение неполного описания типа создает тип. Если неполное описание типа имеет раздел дискриминантов, то это предвыполнение включает предвыполнение раздела дискриминантов: в этом случае раздел дискриминантов полного описания типа не предвыполняется.

Пример рекурсивного типа:

```
type ЯЧЕЙКА; - - неполное описание типа
type СВЯЗЬ is access    ЯЧЕЙКА;
type ЯЧЕЙКА is
  record
     ЗНАЧЕНИЕ: INTEGER;
     СЛЕД: СВЯЗЬ;
     ПРЕД: СВЯЗЬ;
  end record;
ГОЛОВА: СВЯЗЬ: = new ЯЧЕЙКА' (0, null, null);
СЛЕДУЮЩИЙ: СВЯЗЬ: = ГОЛОВА. СЛЕД;
```

Примеры взаимозависимых ссылочных типов:

```
type ПЕРСОНА (ПОЛ: РОД); - - неполное описание типа
type АВТОМОБИЛЬ;              - - неполное описание типа
type ИМЯ_ПЕРСОНЫ is access ПЕРСОНА;
type ИМЯ_АВТОМОБИЛЯ  is access    АВТОМОБИЛЬ;
type АВТОМОБИЛЬ is
  record
     НОМЕР: INTEGER;
     ВЛАДЕЛЕЦ: ИМЯ_ПЕРСОНЫ;
  end record;
type ПЕРСОНА (ПОЛ: РОД) is
  record
     ИМЯ: STRING (1. .20);
     РОДИЛСЯ: ДАТА;
     ВОЗРАСТ: INTEGER range 0. .130;
     МАШИНА: ИМЯ_АВТОМОБИЛЯ;
     case ПОЛ is
     when М = > ЖЕНА: ИМЯ_ПЕРСОНЫ (ПОЛ = > Ж);
     when   Ж = > МУЖ: ИМЯ_ПЕРСОНЫ (ПОЛ = > М);
  end case;
end record;
МОЙ_АВТО, ВАШ_АВТО, СЛЕДУЮЩИЙ_АВТО:
ИМЯ_АВТОМОБИЛЯ; - - неявно инициированы пустым значением
```

3.8.2. Операции над ссылочными типами

Базовые операции над ссылочным типом включают присваивание, генераторы для этого ссылочного типа, проверку принадлежности, явное преобразование, квалификацию и литерал **null**. Если указываемый тип – именуемый тип с дискриминантами, то базовые операции включают именование соответствующих дискриминантов; если указываемый тип – именуемый тип, то они включают именование соответствующих компонентов; если указываемый тип – индексируемый тип, то они включают образование индексированных компонентов и отрезков; если указываемый тип – задачный тип, то они включают именование входов и семейств входов. Кроме того, базовые операции включают образование именованного компонента с зарезервированным словом **all** (см. 4.1.3).

Если указываемый тип – индексируемый тип, то базовые операции включают атрибуты с обозначениями FIRST, LAST, RANGE и LENGTH (и эти же атрибуты с параметром К для номера измерения). Префикс каждого из этих атрибутов должен быть значением ссылочного типа. Эти атрибуты

вырабатывают соответствующие характеристики указываемого объекта (см. 3.6.2).

Если указываемый тип – задачный тип, то базовые операции включают атрибуты с обозначениями TERMINATED и CALLABLE (см. 9.9). Префикс каждого из этих атрибутов должен быть значением ссылочного типа. Эти атрибуты вырабатывают соответствующие характеристики задачных объектов.

Кроме того, атрибут T'BASE (см. 3.3.3) и атрибуты представления T'SIZE и T'STORAGE_SIZE (см. 13.7.2) определены для ссылочного типа или подтипа T; атрибуты A'SIZE и A'ADDRESS определены для ссылочного объекта A (см. 13.7.2).

Кроме базовых, операции над ссылочным типом включают предопределенное сравнение на равенство и неравенство.

3.9. **Разделы описаний**

Раздел описаний содержит элементы описания (возможно и ни одного).

раздел_описаний : : = {основной_элемент_описания}
{дополнительный_элемент_описания}
основной_элемент_описания : : = основное_описание
| спецификатор_представления
| спецификатор_использования
дополнительный_элемент_описания : : = тело
|описание_подпрограммы | описание_пакета
| описание_задачи | описание_настройки
| спецификатор_использования | конкретизация_настройки
тело : : = соответствующее_тело | след_тела
соответствующее_тело : : = тело_подпрограммы
| тело_пакета | тело_задачи

Предвыполнение раздела описаний состоит из предвыполнения элементов описания, если они есть, в порядке их следования в разделе описаний. После своего предвыполнения элемент описания называется *предвыполненным*. До окончания своего предвыполнения (включая фазу перед предвыполнением) элемент описания еще не предвыполнен.

Для нескольких форм элементов описания правила языка (в частности, правило определения области действия и правила видимости) таковы, что использование понятия до предвыполнения элемента описания, который объявляет это понятие, либо невозможно, либо является неправильным. Например, невозможно использование имени типа для описания объекта, если соответствующее описание типа еще не предвыполнено. В случае тел осуществляются следующие проверки:

- При вызове подпрограммы проверяется, предвыполнено ли уже тело подпрограммы.
- При активизации задачи проверяется, предвыполнено ли уже тело соответствующего задачного модуля.
- При конкретизации настраиваемого модуля, имеющего тело, проверяется, предвыполнено ли уже тело настраиваемого модуля.

Если одна из этих проверок дает отрицательный результат, возбуждается исключение PROGRAM_ERROR.

Если описание подпрограммы, описание пакета, описание задачи или описание настраиваемого понятия является элементом описания данного раздела описаний, то тело (если оно существует) программного модуля, описанного элементом описания, должно само быть элементом описания этого раздела описаний (и должно помещаться ниже). Если тело представлено следом тела, то для этого программного модуля необходим раздельно компилируемый субмодуль, содержащий соответствующее тело (см. 10.2).

4. ИМЕНА И ВЫРАЖЕНИЯ

В этой главе приведены правила, применяемые к различным формам имен и выражений, а также при их вычислении.

4.1. **Имена**

Имена могут обозначать понятия, описанные явно или неявно (см. 3.1). Имена могут обозначать также объекты, указанные ссылочными значениями, подкомпоненты и отрезки объектов и значений, одиночные входы, семейства входов и входы семейства входов. Наконец, имена могут обозначать атрибуты этих понятий и объектов.

имя : : = простое_имя
| символьный_литерал | знак_операции
| индексируемый_компонент | отрезок
| именуемый_компонент | атрибут
простое_имя : : = идентификатор
префикс : : = имя | вызов_функции

Простое имя понятия – это либо идентификатор, связанный с этим понятием его описанием, либо другой идентификатор, связанный с этим понятием описанием переименования.

Определенные формы имени (индексируемые и именуемые компоненты, отрезки и атрибуты) включают в себя префикс, который может быть именем или вызовом функции. Если тип префикса – ссылочный тип, то префикс не должен быть именем, которое обозначает формальный параметр вида **out** или его подкомпонент.

Если префикс имени – вызов функции, то имя обозначает компонент, отрезок, вход или семейство входов результата вызова функции либо (если результат – ссылочное значение) объект, указанный результатом.

Говорят, что префикс *соответствует* некоторому типу в любом из следующих случаев:

- Тип префикса – это рассматриваемый тип.
- Тип префикса – ссылочный тип, который указывает на рассматриваемый тип.

Вычисление имени определяет понятие, обозначенное этим именем. Для простого имени, символьного литерала или знака операции вычисление имени не имеет другого результата.

Вычисление имени, имеющего префикс, включает в себя вычисление префикса, т. е. соответствующего имени или вызова функции. Если тип префикса – ссылочный тип, то вычисление префикса включает в себя определение объекта, указанного соответствующим ссылочным значением; если значение префикса является пустым ссылочным значением, исключая случай префикса атрибута представления (см. 13.7.2), то возбуждается исключение CONSTRAINT_ERROR (см. 13.7.2).

Примеры простых имен:

ПИ -- простое имя числа (см. 3.2.2)
ПРЕДЕЛ -- простое имя константы (см. 3.2.1)
СЧЕТ -- простое имя скалярной переменной (см. 3.2.1)
ПУЛЬТ -- простое имя массива (см. 3.6.1)
МАТРИЦА -- простое имя типа (см. 3.6)
СЛУЧАЙНОЕ -- простое имя функции (см. 6.1)
ОШИБКА -- протое имя исключения (см. 11.1)

4.1.1. Индексируемые компоненты

Индексируемый компонент обозначает компонент массива или вход семейства входов.

индексируемый компонент : : =
префикс (выражение {, выражение})

Для компонентов массива тип префикса должен соответствовать индексируемому типу. Значения индексов компонента задаются выражениями, каждой позиции индекса должно соответствовать одно такое выражение. Для входа из семейства входов префикс должен быть именем, которое обозначает семейство входов задачного объекта, а выражение (оно должно быть только одно) задает значения индекса конкретного входа.

Типом каждого выражения должен быть тип соответствующего индекса. В языке не определяется порядок вычисления префикса и выражений при вычислении индексируемого компонента. Если значение индекса не принадлежит диапазону индекса массива или семейства входов, определяемых префиксом, то возбуждается исключение CONSTRAINT_ERROR.

Примеры индексируемых компонентов:

МОЕ_РАСПИСАНИЕ (СБ) -- компонент одномерного массива (см. 3.6.1)
СТРАНИЦА (10) -- компонент одномерного массива (см. 3.6)
ПУЛЬТ (М, К+1) -- компонент двумерного массива (см. 3.6.1)
СТРАНИЦА (10) (20) -- компонент компонента (см. 3.6)
ЗАПРОС (СРЕДА) -- вход семейства входов (см. 9.5)
СЛЕДУЮЩЕЕ_ОБРАМЛЕНИЕ (Л), (М,Н) -- компонент вызова функции (см. 6.1)

Примечание к примеру. Для компонентов многомерных массивов (таких как ПУЛЬТ) и массива массивов (таких как СТРАНИЦА) используются различные обозначения. Компонентами массива массивов являются массивы, и они могут быть индексированы. Так, СТРАНИЦА (10) (20) обозначает двадцатый компонент массива СТРАНИЦ (10). В последнем примере СЛЕДУЮЩЕЕ_ОБРАМЛЕНИЕ (Л) – вызов функции, возвращающий ссылочное значение, указывающее двумерный массив.

4.1.2. Отрезки

Отрезок обозначает одномерный массив из нескольких последовательных компонентов одномерного массива. Отрезок переменной – переменная, отрезок константы – константа, отрезок значения – значение.

отрезок : : = префикс (дискретный_диапазон)

Префикс отрезка должен соответствовать одномерному индексируемому типу. Тип отрезка – это базовый тип этого индексируемого типа. Границы дискретного диапазона определяют границы отрезка и должны быть того же типа, что и тип индекса; отрезок является *пустым*, т. е. обозначает пустой массив, если дискретный диапазон является пустым.

Порядок вычисления префикса и дискретного диапазона при вычислении имени отрезка в языке не определяется. Если при вычислении отрезка хотя бы одна из границ дискретного диапазона не принадлежит диапазону индексов, определяемому префиксом отрезка, то (кроме случая пустого отрезка) возбуждается исключение CONSTRAINT_ERROR. (Границы пустого отрезка могут не принадлежать подтипу индекса.)

Примеры отрезков:

```
ЗВЕЗДОЧКИ (1. .15) - - отрезок из 15 символов (см. 3.6.3)
СТРАНИЦА (10. .10+РАЗМЕР)
- - отрезок из 1+РАЗМЕР компонентов (см. 3.6 и 3.2.1)
СТРАНИЦА (Л) (А. .В) - - отрезок массива СТРАНИЦА (Л) (см. 3.6)
ЗВЕЗДОЧКИ (1. .0) - - пустой отрезок (см. 3.6.3)
МОЕ_РАСПИСАНИЕ (ДЕНЬ НЕДЕЛИ)
- - границы задаются подтипом (см. 3.6 и 3.5.1)
ЗВЕЗДОЧКИ (5. .15) (К)
- - то же, что ЗВЕЗДОЧКИ (К), если К в диапазоне 5. .15 (см. 3.6)
```

Примечание. Для одномерного массива А имя А (К. .К) задает отрезок, состоящий из одного компонента; его тип соответствует базовому типу массива А. С другой стороны, А (К) – компонент массива и имеет тип соответствующей компоненты.

4.1.3. Именуемые компоненты

Именуемые компоненты используются для обозначения компонентов записей, входов, семейств входов и объектов, указанных ссылочными значениями; они используются также в качестве *расширенных имен*, как это описано ниже.

именуемый_компонент: : = префикс. постфикс

постфикс : : = простое_имя | символьный_литерал

| знак_операции | all

Для обозначения дискриминанта, компонента записи, входа или объекта, указанного ссылочным значением, используются следующие четыре формы именуемых компонентов.

а) Дискриминант.

Постфикс должен быть простым именем, обозначающим дискриминант объекта или значения. Префикс должен соответствовать типу этого объекта или значения.

б) Компонент записи.

Постфикс должен быть простым именем, обозначающим компонент именуемого объекта или значения. Префикс должен соответствовать типу этого объекта или значения.

Для компонентов варианта делается проверка: являются ли значения дискриминантов такими, что запись имеет этот компонент. В противном случае возбуждается исключение CONSTRAINT_ERROR.

в) Одиночный вход или семейство входов задачи.

Постфикс должен быть простым именем, обозначающим одиночный вход или семейство входов задачи. Префикс должен соответствовать типу этой задачи.

г) Объект, указанный ссылочным значением.

Постфикс должен быть зарезервированным словом **all**. Значение префикса должно принадлежать ссылочному типу.

Именуемый компонент одной из двух нижеуказанных форм называется *расширенным именем*. В каждом случае постфикс должен быть либо простым именем, либо символьным литералом, либо знаком операции. Вызов функции в качестве префикса расширенного имени не допускается. Расширенное имя может обозначать:

д) Понятие, описанное в видимом разделе описания пакета.

Префикс должен обозначать пакет. Постфикс должен быть простым именем, символьным литералом или знаком операции понятия.

е) Понятие, описание которого находится непосредственно в поименованной конструкции.

Префикс должен обозначать одну из следующих конструкций: программный модуль, оператор блока, оператор цикла или оператор принятия. Для оператора принятия префикс должен быть либо простым именем входа или семейства входов, либо расширенным именем, заканчивающимся таким простым именем (т. е. не допускается индекс). Постфикс должен быть простым именем, символьным литералом или знаком операции такого понятия, чье описание находится непосредственно в конструкции.

Данная форма расширенного имени допустима только в самой конструкции (включая тело и любые субмодули в случае программного модуля). Не допускается использование в качестве префикса имен, описанных с помощью описания переименования. Если префикс – это имя подпрограммы или оператора принятия и если существует более одной видимой объемлющей подпрограммы или оператора принятия с таким именем, то расширенное имя неопределенно, независимо от постфикса.

Если в соответствии с правилами видимости возможна по крайней мере одна интерпретация префикса именуемого компонента как имени объемлющей подпрограммы или оператора принятия, то рассматриваются только те интерпретации, которые соответствуют правилу *е* – т. е. расширенные имена. (Интерпретация префикса как вызова функции не рассматривается).

Вычисление имени, являющегося именуемым компонентом, включает вычисление префикса.

Примеры именуемых компонентов:

ЗАВТРА.МЕСЯЦ - - компонент записи (см. 3.7)

СЛЕДУЮЩИЙ_АВТО.ВЛАДЕЛЕЦ - - компонент записи (см. 3.8.1)

СЛЕДУЮЩИЙ_АВТО.ВЛАДЕЛЕЦ.ВОЗРАСТ - - компонент записи (см. 3.8.1)

ПИШУЩЕЕ_УСТРОЙСТВО.МОДУЛЬ - - компонент записи (дискриминант) (см. - - компонент записи (дискриминант) (см. 3.7.3)

МИН_ЯЧЕЙКА (Н).ЗНАЧЕНИЕ - - компонент записи результата
- - вызова функции (см. 6.1 и 3.8.1)
УПРАВЛЕНИЕ.ЗАХВАТИТЬ - - вход задачи УПРАВЛЕНИЕ (см. 9.1 и 9.2)
ПУЛ(К).ПИСАТЬ - - вход задачи ПУЛ(К) (см. 9.1 и 9.2)
СЛЕДУЮЩИЙ_АВТО.all - - объект, указанный ссылочной перемен-
- - ной СЛЕДУЮЩИЙ_АВТО (см. 3.8.1)

Примеры расширенных имен:

УПРАВЛЕНИЕ_ТАБЛИЦЕЙ.ВНЕСТИ
- - процедура видимого раздела пакета (см. 7.5)
УПРАВЛЕНИЕ_ПО_КЛЮЧУ."<" - - операция видимого раздела пакета (см. 7.4.2)
СКАЛ_ПРОИЗВЕДЕНИЕ.СУММА
- - переменная, описанная в теле процедуры (см. 6.5)
БУФЕР.ПУЛ - - переменная, описанная в задачном модуле (см. 9.12)
БУФЕР.ЧИТАТЬ - - вход задачного модуля (см. 9.12)
ОБМЕН.РАБОЧАЯ_ЯЧЕЙКА - - переменная оператора блока (см 5.6)
STANDARD BOOLEAN - - имя предопределенного типа (см. 8.6
- - и обязательное приложение 3)

Примечание. Для записей, компонентами которых являются другие записи, перечисленные правила означают, что простое имя должно быть дано для каждого уровня имени подкомпонента. Например, имя СЛЕДУЮЩИЙ_АВТО.ВЛАДЕЛЕЦ.ВОЗРАСТ.МЕСЯЦ не может быть укорочено (СЛЕДУЮЩИЙ_АВТО.ВЛАДЕЛЕЦ.МЕСЯЦ недопустимо).

4.1.4. Атрибуты

Атрибут обозначает базовую операцию над понятием, задаваемым префиксом.

атрибут : : = префикс'обозначение_атрибута

обозначение_атрибута : : =

простое_имя [(*универсальное_статическое_*выражение)]

Применимые обозначения атрибутов зависят от конкретного префикса. Атрибут может быть базовой операцией, вырабатывающей значение, но может быть и функцией, типом или диапазоном. Смысл префикса атрибута должен быть определимым независимо от обозначения атрибута и независимо от того, что это есть префикс именно атрибута.

Определенные в языке атрибуты приведены в обязательном приложении 1. Конкретная реализация может ввести дополнительные атрибуты, описание которых должно быть дано в руководстве по реализации в соответствии с обязательным приложением 4. Обозначения таких атрибутов должны отличаться от обозначений атрибутов, определенных в языке.

Вычисление имени, являющегося атрибутом, состоит из вычисления префикса.

Примечание. Обозначения атрибутов DIGITS, DELTA и RANGE имеют идентификаторы, совпадающие с зарезервированными словами. Однако неоднозначности нет, поскольку перед обозначением атрибута всегда стоит апостроф. Единственными обозначениями предопределенных атрибутов, содержащими универсальное выражение, являются те, которые соответствуют определенным операциям над индексируемыми типами (см. 3.6.2).

Примеры атрибутов:

ЦВЕТ'FIRST - - минимальное значение перечислимого типа ЦВЕТ (см. 3.3.1 и 3.5)

РАДУГА'BASE'FIRST - - то же, что и атрибут ЦВЕТ'FIRST (см. 3.3.2 и 3.3.3)
ВЕЩЕСТВ'DIGITS - - точность типа ВЕЩЕСТВ (см. 3.5.7 и 3.5.8)
ПУЛЬТ'LAST (2)
- - верхняя граница диапазона 2-го индекса для ПУЛЬТ (см. 3.6.1 и 3.6.2)
ПУЛЬТ'RANGE (1) - - диапазон первого индекса для ПУЛЬТ (см. 3.6.1 и 3.6.2)
ПУЛ (К) 'TERMINATED - - TRUE, если задача ПУЛ (К) завершена (см. 9.2 и 9.9)
ДАТА'SIZE - - количество битов для записи типа ДАТА (см. 3.7 и 13.7.2)
СООБЩЕНИЕ'ADDRESS
- - адрес переменной СООБЩЕНИЕ именуемого типа (см. 3.7.2 и 13.7.2)

4.2. **Литералы**

Литерал — это либо числовой литерал, либо литерал перечисления, либо литерал **null**, либо строковый литерал, либо символьный литерал. Вычисление литерала вырабатывает соответствующее значение.

Числовые литералы — это литералы типов *универсальный_целый* и *универсальный_вещественный.* Литералы перечисления включают символьные литералы и вырабатывают значения соответствующих перечислимых типов. Литерал **null** вырабатывает пустое ссылочное значение, которое не указывает ни на какой объект вообще.

Строковый литерал — это базовая операция, которая преобразует последовательность символов в значение одномерного массива с компонентами символьного типа; границы этого массива определяются в соответствии с правилами для позиционных агрегатов массива (см. 4.3.2). Для пустого строкового литерала верхняя граница массива является предшественником нижней границы, выдаваемой атрибутом PRED. Вычисление пустого строкового литерала возбуждает исключение CONSTRAINT_ERROR, если нижняя граница не имеет предшественника (см. 3.5.5).

Тип строкового литерала и тип литерала **null** должны определяться исключительно из контекста, в котором эти литералы встречаются, без учета самого литерала, используя при этом только тот факт, что литерал **null** — это значение ссылочного типа, а строковый литерал — значение одномерного массива, тип компонентов которого — символьный.

Символьные литералы, содержащиеся в строковом литерале и соответствующие графическим символам, должны быть видимы в месте нахождения строкового литерала (хотя сами эти символы для определения типа данного строкового литерала не используются).

Примеры:

3.14159_26536	- - литерал вещественного типа
1_345	- - литерал целого типа
ТРЕФЫ	- - литерал перечислений
'A'	- - символьный литерал
„НЕКОТОРЫЙ_ТЕКСТ"	- - строковый литерал

4.3. **Агрегаты**

Агрегат — это базовая операция, которая объединяет значения компонентов в составное значение именуемого или индексируемого типа.

агрегат : : = (сопоставление_компонентов
{ , сопоставление_компонентов })

сопоставление_компонентов : : = [выбор {| выбор} = >] выражение

Каждое сопоставление компонентов связывает выражение с компонентами. Сопоставление компонентов называется *именованным*, если компоненты явно определены выборами, и *позиционным* в противном случае. При позиционном сопоставлении каждому отдельно взятому компоненту неявно соответствует некоторая позиция: именуемым компонентам — в порядке следования их описаний, индексируемым компонентам — по возрастанию индекса.

Именованные сопоставления могут стоять в произвольном порядке (исключая выбор **others**), но если в агрегате одновременно используются позиционные и именованные сопоставления, то первыми должны стоять позиционные сопоставления, каждое на своем месте. Следовательно, за именованными сопоставлениями в агрегате должны следовать только именованные сопоставления. В агрегатах, содержащих единственное сопоставление, должно всегда использоваться именованное сопоставление. Правила для сопоставления компонентов агрегатов именуемого типа и агрегатов индексируемого типа определены в разд. 4.3.1 и 4.3.2.

Синтаксис выборов сопоставления компонентов совпадает с синтаксисом разделов вариантов (см. 3.7.3). Выбор, являющийся простым именем компонента, допустим только в агрегатах именуемого типа. Выбор, являющийся простым выражением или дискретным диапазоном, допустим только в агрегатах индексируемого типа; выбор, являющийся простым выражением, задает значение индекса; дискретный диапазон задает диапазон значений индекса. Выбор **others** допустим только в последнем сопоставлении компонентов в качестве единственного выбора и определяет все оставшиеся компоненты, если они есть.

Каждый компонент значения, определяемого агрегатом, должен встретиться в агрегате один и только один раз. Следовательно, каждый агрегат должен быть полным и не допускается, чтобы данный компонент был задан более чем одним выбором.

Тип агрегата должен быть определимым исключительно по контексту, в котором встречается агрегат, без учета самого агрегата, используя только тот факт, что его тип должен быть составным и не лимитируемым. Тип агрегата в свою очередь определяет требуемый тип каждого его компонента.

Примечание. Приведенное выше правило означает, что для определения типа агрегата не может быть использована информация, которую несет в себе агрегат. В частности, это определение не может использовать тип выражения в сопоставлении компонентов, формы или типы выборов. Агрегат с одним компонентом всегда можно отличить от выражения, заключенного в скобки, благодаря обязательному именованию компонента такого агрегата.

4.3.1. Агрегаты записей

Для агрегата именуемого типа (агрегата записи) имена компонентов, заданные выборами, должны обозначать компоненты (включая дискриминанты) именуемого типа. Выбор **others** в агрегатах записей должен представлять хотя бы один компонент. Сопоставление компонентов с выбором **others** или более чем с одним выбором допускается только тогда, когда

представленные компоненты имеют один и тот же тип. Выражение в сопоставлении компонентов должно иметь тип соответствующего компонента записи.

Значение, определяющее дискриминант, должно быть задано статическим выражением (заметим, что это значение определяет, какие из зависимых компонентов должны присутствовать в значении записи).

При вычислении агрегатов записи порядок вычисления выражений в сопоставлениях компонентов в языке не определен. Выражение в именованном сопоставлении вычисляется один раз для каждого сопоставленного компонента. Производится проверка на принадлежность значения каждого подкомпонента агрегата подтипу этого подкомпонента. При отрицательном результате проверки возбуждается исключение CONSTRAINT_ERROR.

Пример агрегата записи с позиционным сопоставлением:

(4, ИЮЛЬ, 1776) - - см. 3.7

Примеры агрегатов записи с именованными сопоставлениями:

(ДЕНЬ => 4, МЕСЯЦ => ИЮЛЬ, ГОД => 1776)
(МЕСЯЦ => ИЮЛЬ, ДЕНЬ => 4, ГОД => 1776)
(ДИСК, ЗАКРЫТ, ТРАКТ = > 5, ЦИЛИНДР => 12) - - см. 3.7.3
(УСТРОЙСТВО => ДИСК, СОСТОЯНИЕ => ЗАКРЫТ, ЦИЛИНДР => 9, ТРАКТ => 1)

Примеры сопоставления компонентов с несколькими выборами:

(ЗНАЧЕНИЕ => 0, СЛЕД | ПРЕД => new ЯЧЕЙКА'(0, null, null)) - - см. 3.8.1
- - генератор вычисляется дважды: СЛЕД и ПРЕД обозначают разные ячейки

Примечание. В агрегате с позиционными сопоставлениями первыми идут значения дискриминантов, так как раздел дискриминантов идет первым в описании именуемого типа; они должны быть в том же порядке, что и в разделе дискриминантов.

4.3.2. Агрегаты массивов

Если тип агрегата – одномерный индексируемый тип, то каждый выбор должен задавать значения индекса, а выражение в каждом сопоставлении компонентов должно иметь тип соответствующего компонента.

Если тип агрегата – многомерный индексируемый тип, то К-мерный агрегат записывается как одномерный, в котором выражения, задающие сопоставления компонентов, сами записываются как (К – 1)-мерный агрегат, называемый *подагрегатом*; подтип индекса одномерного агрегата задается первой позицией индекса индексируемого типа. То же правило используется для следующей позиции индекса при записи подагрегатов, если они опять многомерные. В многомерном агрегате допустимо использование строкового литерала в качестве одномерного массива с компонентами символьного типа. В дальнейшем связанные с агрегатами массивов правила формулируются в терминах одномерных агрегатов.

За исключением последнего сопоставления компонентов с единственным выбором others остальные сопоставления (если они есть) агрегата массива должны быть либо все позиционными, либо все именованными. Для агрегата массива, имеющего одно именованное сопоставление компонентов с одним выбором, допускается задание только такого выбора, который не является статическим или является пустым диапазоном. Выбор others явля-

ется статическим, если статическим является соответствующее ограничение индекса.

Границы агрегата массива, имеющего выбор others, определяются соответствующим ограничением индекса. Использование выбора others допускается только тогда, когда агрегат находится в одном из следующих контекстов (контекст определяет соответствующее ограничение индекса).

а) Агрегат – это фактический параметр, фактический параметр настройки, выражение, являющееся результатом функции, или выражение, которое следует за составным ограничителем присваивания, и подтип соответствующего формального параметра, формального параметра настройки, результата функции или объекта – ограниченный индексируемый подтип.

Для агрегата, помещенного в такой контекст и содержащего сопоставление с выбором others, другие именованные сопоставления допускаются только в случае фактического параметра (не являющегося фактическим параметром настройки) или результата функции. Если агрегат – многомерный массив, то это требование распространяется и на все его подагрегаты.

б) Агрегат – это операнд квалифицированного выражения, обозначение типа которого указывает ограниченный индексируемый подтип.

в) Агрегат – это выражение в сопоставлении компонентов другого агрегата индексируемого или именуемого типа и, если этот объемлющий агрегат – многомерный агрегат индексируемого типа, то сам он заключен в один из этих трех видов контекста.

Границы агрегата массива без выбора others определяются следующим образом. Для агрегата с именованными сопоставлениями границы определяются наименьшим и наибольшим из заданных выборов. Нижняя граница позиционного агрегата определяется соответствующим ограничением индекса, если агрегат помещен в один из контекстов *а, б* или *в*; в противном случае нижняя граница задается как И'FIRST, где И – подтип индекса; в обоих случаях верхняя граница определяется числом компонентов.

Вычисление агрегата массива, не являющегося подагрегатом, производится в два шага. На первом шаге вычисляются выборы данного агрегата и его подагрегатов (если они есть) в порядке, не определенном в языке. На втором – вычисляются выражения в сопоставлениях компонентов в порядке, также не определенном в языке. Выражения в именованном сопоставлении вычисляются один раз для каждого сопоставляемого компонента. Вычисление подагрегатов состоит из этого второго шага (первый шаг пропускается, так как выборы уже были вычислены).

При вычислении не пустого агрегата массива производится проверка того, что значения задаваемых выборами индексов принадлежат соответствующему подтипу индекса, а также того, что значение каждого подкомпонента агрегата принадлежит подтипу этого подкомпонента. Для К-мерного агрегата производится проверка того, что все (К–1)-мерные подагрегаты имеют одинаковые границы. Если бы хотя бы одна из этих проверок дает отрицательный результат, возбуждается исключение CONSTRAINT_ERROR.

Примечание. Допустимыми для агрегата массива с выбором others являются те контексты, в которых границы такого агрегата всегда известны.

Примеры агрегатов массивов с позиционными сопоставлениями:

```
(7, 9, 5, 1, 3, 2, 4, 8, 6, 0)
ТАБЛИЦА'(5, 8, 4, 1, others => 0) -- см. 3.6
```

Примеры агрегатов массивов с именованными сопоставлениями:

```
(1..5 => (1..8 => 0.0)) -- двумерный
(1..К => new ЯЧЕЙКА) -- К новых ячеек, в частности, для К = 0
ТАБЛИЦА'(2|4|10 => 1, others => 0)
РАСПИСАНИЕ'(ПНД..ПТН => TRUE, others => FALSE) -- см. 3.6
РАСПИСАНИЕ'(СРД|ВСК => FALSE, others => TRUE)
```

Примеры агрегатов двумерных массивов:

```
-- три агрегата для одного значения типа МАТРИЦА, см. 3.6
((1.1, 1.2, 1.3), (2.1, 2.2, 2.3))
(1 => (1.1, 1.2, 1.3), 2=> (2.1, 2.2, 2.3))
(1 => (1 => 1.1, 2=> 1.2, 3=> 1.3),
2=> (1 => 2.1, 2 => 2.2, 3 => 2.3))
```

Примеры агрегатов в качестве значений инициализации:

```
A: ТАБЛИЦА := (7, 9, 5, 1, 3, 2, 4, 8, 6, 0); -- A(1) = 7, A(10) = 0
B: ТАБЛИЦА: = ТАБЛИЦА'(2|4|10 => 1, others => 0); -- B(1) = 0, B(10) = 1;
C: constant МАТРИЦА: = (1..5 => (1..8 => 0.0)); -- C'FIRST(1) = 1, C'LAST(2) = 8

D: ВЕКТОР_БИТ (М..К) : = (М..К => TRUE); -- см. 3.6
E: ВЕКТОР_БИТ (М..К) : = (others => TRUE);
F: STRING (1..1) : = (1 => 'F'); -- однокомпонентный агрегат; то же, что и „F"
```

4.4. **Выражения**

Выражение — это формула, которая определяет способ вычисления значения.

```
выражение : : = отношение {and отношение}
| отношение {and then отношение}|отношение{or отношение}
| отношение {or else отношение}| отношение {xor отношение}
отношение : : =
простое_выражение [операция_отношения простое_выражение]
| простое_выражение [not] in диапазон
| простое_выражение [not] in обозначение_типа
простое_выражение : : =
[унарная_аддитивная_операция] слагаемое
{бинарная_аддитивная_операция слагаемое}
слагаемое : : =
множитель {мультипликативная_операция множитель}
множитель : : = первичное [** первичное]
| abs первичное | not первичное
первичное : : = числовой_литерал | null | агрегат
| строковый_литерал | имя | генератор
| вызов_функции | преобразование_типа
| квалифицированное_выражение | (выражение)
```

Каждое первичное имеет значение и тип. Использование имен в качестве первичного допускается только для именованных чисел, атрибутов, кото-

рые вырабатывают значения, а также имен, обозначающих объекты (значением такого первичного является значение объекта) или обозначающих значения. Не допускается в качестве первичных использование имен формальных параметров вида **out**, а использование имен их подкомпонентов допускается только в случае дискриминантов.

Тип выражения зависит только от типа его составных частей и применяемых операций; для совмещенных операндов или операций определение типа операнда или идентификация операции зависит от контекста. Для каждой предопределенной операции типы операндов и результата приведены в разд. 4.5.

Примеры первичных:

```
4.0                  - - литерал вещественного типа
ПИ                   - - именованное число
(1..10 => 0)         - - агрегат индексируемого типа
СУММА                - - переменная
INTEGER'LAST - - атрибут
СИНУС (X)            - - вызов функции
ЦВЕТ'(СИНИЙ) - - квалифицированное выражение
ВЕЩЕСТВ (М*К) - - преобразование типа
(СЧЕТ_СТРОЧЕК +10) - - выражение в скобках
```

Примеры выражений:

```
ТОМ                                    - - первичное
not ИСПОРЧЕН                           - - множитель
2 * СЧЕТ_СТРОЧЕК                       - - слагаемое
–4.0                                   - - простое выражение
–4.0 + A                               - - простое выражение
B ** 2–4.0 * A * C                     - - простое выражение
ПАРОЛЬ (1..3) = "АБВ"                  - - отношение
СЧЕТ in МАЛОЕ_ЦЕЛ                      - - отношение
СЧЕТ not in МАЛОЕ_ЦЕЛ                  - - отношение
ИНДЕКС = 0 or ЭЛЕМЕНТ_ГР               - - выражение
(ХОЛОД and СОЛНЦЕ) or ТЕПЛО            - - выражение (скобки обязательны)
A **(B ** C)                           - - выражение (скобки обязательны)
```

4.5. **Операции и вычисление выражения**

В языке определены шесть классов операций. При описании функций, определяющих пользовательские операции, в качестве обозначений могут быть использованы приведенные ниже знаки операций (исключая /=). Шесть классов операций приведены в порядке возрастания их старшинства.

```
логическая_операция : : = and | or | xor
операция_отношения : : = = | / = | < | <= | > | > =
бинарная_аддитивная_операция : : = + | – | &
унарная_аддитивная_операция : : = + | –
мультипликативная_операция : : = * | / | mod | rem
операция_высшего_приоритета : : = ** | abs | not
```

Формы управления с промежуточной проверкой **and then** и **or else** имеют тот же порядок старшинства, что и логические операции. Проверки принадлежности **in** и **not in** имеют то же старшинство, что и операции отношения.

В слагаемом, простом выражении, отношении или выражении, группирование операций с их операндами проводится сначала для операций с боль-

шим старшинством, а затем для операций с меньшим старшинством. В случае последовательных операций с одинаковым старшинством группирование операций с их операндами производится в порядке их текстуального следования слева направо; для изменения порядка группирования могут использоваться скобки.

В языке не определяется порядок вычисления операндов множителя, слагаемого, простого выражения или отношения и операндов выражения, которое не содержит форм управления с промежуточной проверкой (но вычисление операндов производится до применения соответствующей операции). Правый операнд формы управления с промежуточной проверкой вычисляется тогда и только тогда, когда левый операнд имеет определенное значение (см. 4.5.1).

Для каждой формы описания типа некоторые из перечисленных операций являются *предопределенными*, т. е. неявно задаются описанием типа. Для каждого такого неявного описания операции именами параметров являются LEFT и RIGHT для бинарных операций; для унарных аддитивных операций и унарных операций **abs** и **not** их единственный параметр именуется RIGHT. В разд. 4.5.1 – 4.5.7 поясняются результаты предопределенных операций.

Предоиределенные операции над целыми типами либо вырабатывают математически корректный результат, либо возбуждают исключение NUMERIC_ERROR. Предопределенная операция, вырабатывающая результат целого типа (отличного от *универсального_целого*), может лишь возбуждать исключение NUMERIC_ERROR, только если математический результат не является значением этого целого типа. Предопределенные операции над вещественными типами вырабатывают результаты, точность которых определяется в соответствии с разд. 4.5.7. Предопределенная операция, вырабатывающая результат вещественного типа (отличного от *универсального_вещественного*), может возбуждать исключение NUMERIC_ERROR, только если ее результат не принадлежит диапазону хранимых чисел этого типа, как это поясняется в разд. 4.5.7.

Примеры старшинства

```
not СОЛНЦЕ or ТЕПЛО -- совпадает с (not СОЛНЦЕ) or ТЕПЛО
X > 4.0 and Y > 0.0 -- совпадает с (X > 4.0) and (Y > 0.0)
–4.0 * A ** 2       -- совпадает с – (4.0 * (A ** 2))
abs (1 + A) + B     -- совпадает с (abs (1 + A)) + B
Y ** (–3)           -- скобки необходимы
A/B * C             -- совпадает с (A/B) * C
A + (B + C)         -- вычисляется B + C, а затем к результату прибавляется A
```

4.5.1. Логические операции и формы управления с промежуточной проверкой

Приводимые в табл. 4.1 логические операции предопределены для любых логических типов и любых одномерных индексируемых типов с компонентами логического типа. В обоих случаях операнды должны иметь один и тот же тип.

Таблица 4.1

Знак операции	*Операция*	*Тип операнда*	*Тип результата*
and	Конъюнкция	Любой логический тип Массив логических компонентов	Тот же логический тип Тот же индексируемый тип
or	Дизъюнкция	Любой логический тип Массив логических компонентов	Тот же логический тип Тот же индексируемый тип
xor	Исключающая дизъюнкция	Любой логический тип Массив логических компонентов	Тот же логический тип Тот же индексируемый тип

Операции над массивами выполняются покомпонентно, если компоненты есть (как для равенства, см. 4.5.2). Границы массива-результата совпадают с границами левого операнда. Для каждого компонента левого операнда проверяется наличие сопоставленного компонента правого операнда, и наоборот. При нарушении этого правила возбуждается исключение CONSTRAINT_ERROR.

Формы управления с промежуточной проверкой **and then** и **or else** определены для двух операндов логического типа и вырабатывают результат того же самого типа. Левый операнд формы управления с промежуточной проверкой всегда вычисляется первым. Если левый операнд выражения с формой **and then** дает значение FALSE, то правый операнд не вычисляется, и значением выражения является FALSE. Если левый операнд выражения с формой **or else** дает TRUE, то правый операнд не вычисляется, и значением выражения является TRUE. Если вычисляются оба операнда, то результат **and then** такой же, как **and**, а результат **or else** – как **or**.

Примечание. Обычный смысл логических операций задается таблицей истинности (см. табл. 4.2).

Таблица 4.2

A	B	A and B	A or B	A xor B
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

Примеры логических операций:

```
СОЛНЦЕ or ТЕПЛО
ФИЛЬТР (1..10) and ФИЛЬТР (15..24) - - см. 3.6.1
```

Примеры форм управления с промежуточной проверкой:

```
СЛЕДУЮЩИЙ_АВТО.ВЛАДЕЛЕЦ/ = null and then
СЛЕДУЮЩИЙ_АВТО.ВЛАДЕЛЕЦ.ВОЗРАСТ > 25 - - см. 3.8.1
K = 0 or else A (K) = ВЕРХ_ЗНАЧЕНИЕ
```

4.5.2. Операции отношения и проверки принадлежности

Операции равенство и неравенство предопределены для любого, не являющегося лимитируемым, типа. Остальные операции отношения являются операциями упорядочивания: < (меньше), <= (меньше или равно), > (больше), >= (больше или равно). Все эти операции сведены в табл. 4.3. Операции упорядочивания предопределены для любого скалярного типа и любого дискретного индексируемого типа (одномерного индексируемого типа с компонентами дискретного типа). Операнды каждой предопределенной операции отношения имеют один и тот же тип. Тип результата—предопределенный тип BOOLEAN.

Смысл операций отношения традиционный: результат равен TRUE, если соответствующее отношение удовлетворено, результат равен FALSE в противном случае. Операция неравенства дает результат противоположный результату операции равенства: FALSE, если операнды равны, TRUE, если – не равны.

Таблица 4.3

Знак операции	*Операция*	*Тип операнда*	*Тип результата*
= / =	Равенство и неравенство	Любой тип	BOOLEAN
<= >=	Проверка упорядоченности	Любой скалярный тип	BOOLEAN
> <		Дискретный индексируемый тип	BOOLEAN

Равенство для дискретных типов – это равенство значений. Результаты предопределенных операций отношения для вещественных операндов, значения которых равны *приблизительно,* приводятся в разд. 4.5.7. Два ссылочных значения равны, когда они указывают на один и тот же объект, либо когда они равны пустому значению ссылочного типа.

Для двух значений одного и того же индексируемого типа или именуемого типа левый операнд равен правому, если и только если для каждого компонента левого операнда имеется *сопоставленный компонент* правого операнда, и наоборот; значения сопоставленных компонентов равны в смысле предопределенной операции равенства для типа этих компонентов. В частности, всегда равны два пустых массива одного и того же типа, всегда равны две пустых записи одного и того же типа.

Для сравнения двух записей одинакового типа *сопоставленными компонентами* являются компоненты (если они есть) с одинаковыми идентификаторами компонентов.

При сравнении двух одномерных массивов одинакового типа сопоставленными являются компоненты (если они есть), значения индексов которых сопоставляются друг с другом следующим образом: сопоставляются

нижние границы диапазонов индексов друг с другом, следующие за сопоставленными индексами также сопоставляются. При сравнении двух многомерных массивов сопоставленными являются компоненты, у которых значения индексов сопоставлены в порядке позиций индексов.

Если равенство явно определено для лимитируемого типа, оно не распространяется на составные типы, имеющие подкомпоненты лимитируемого типа (допускается явное определение равенства для таких составных типов).

Операции упорядочивания <, <=, > и >=, которые определены для дискретных индексируемых типов, соответствуют *лексикографическому* порядку, основанному на предопределенном отношении порядка для типов компонентов. Пустой массив лексикографически меньше массива, имеющего по крайней мере один компонент. Для непустых массивов левый операнд лексикографически меньше правого операнда, если первый компонент левого операнда меньше первого компонента правого операнда или если их первые компоненты равны, а хвост левого операнда лексикографически меньше правого (хвост состоит из оставшихся компонентов, исключая первый, и может быть пустым).

Проверки принадлежности **in** и **not in** предопределены для всех типов. Типом результата является предопределенный тип BOOLEAN. Для проверки принадлежности диапазону простое выражение и границы диапазона должны быть одного и того же скалярного типа; для проверки принадлежности с обозначением типа тип простого выражения должен быть базовым для этого обозначения. Вычисление проверок принадлежности **in** вырабатывает результат TRUE, если значение простого выражения принадлежит данному диапазону или это значение принадлежит подтипу, заданному обозначением типа; в противном случае вычисление вырабатывает результат FALSE (для значений вещественного типа см. 4.5.7). Проверка принадлежности **not in** вырабатывает результат, противоположный результату проверки принадлежности **in**.

Примеры:

X/ = Y

""<"A" and "A" < "AA" - - TRUE

"AA" < "B" and "A" < "A" - - TRUE

МОЙ_АВТО = null - - истина, если МОЙ_АВТО пуст (см. 3.8.1)

МОЙ_АВТО = ВАШ_АВТО - - истина, если используется один и тот же автомобиль

МОЙ_АВТО. all = ВАШ_АВТО all - - истина, если оба автомобиля идентичны

К not in 1. .10 - - проверка принадлежности диапазону

СЕГОДНЯ in ПНД. .ПТН - - проверка принадлежности диапазону

СЕГОДНЯ in ДЕНЬНЕДЕЛИ - - проверка принадлежности подтипу (см. 3.5.1)

АРХИВ in УСТРОЙСТВО_ДИСКА - - проверка принадлежности подтипу (см. 3.7.3)

Примечание. Предопределенные операции отношения и проверка принадлежности не возбуждает исключений, но исключение может быть возбуждено при вычислении операндов этих операций.

Если именуемый тип имеет компоненты, зависящие от дискриминантов, то компоненты двух значений сопоставлены тогда и только тогда, когда равны их дискриминанты. Компоненты непустых массивов сопоставлены, если и только если значение атрибута LENGTH(K) для каждой позиции индекса одинаковы у обоих массивов.

4.5.3. Бинарные аддитивные операции

Бинарные аддитивные операции + и – предопределены для любого числового типа и имеют свой обычный смысл. Операция катенации & предопределена для любого одномерного нелимитируемого индексируемого типа. Характеристики операций сведены в табл. 4.4.

Таблица 4.4

Знак операции	*Операция*	*Тип левого операнда*	*Тип правого операнда*	*Тип результата*
+	Сложение	Любой числовой тип	Тот же числовой тип	Тот же числовой тип
–	Вычитание	Любой числовой тип	Тот же числовой тип	Тот же числовой тип
&	Катенация	Любой индексируемый тип	Тот же индексируемый тип	Тот же индексируемый тип
		Любой индексируемый тип	Тип компонента	Тот же индексируемый тип
		Тип компонента	Любой индексируемый тип	Тот же индексируемый тип
		Тип компонента	Тип компонента	Любой индексируемый тип

Для вещественных типов точность результата определяется типом операндов (см. 4.5.7).

Если оба операнда – одномерные массивы, то результатом катенации является одномерный массив, длина которого равна сумме длин операндов, а его компоненты составлены из компонентов левого операнда и следующих за ними компонентов правого операнда. Нижняя граница этого результата совпадает с нижней границей левого операнда, если левый операнд – не пустой массив; в противном случае результатом катенации является правый операнд.

Если любой из операндов имеет тип компонента индексируемого типа, то результат катенации определяется по приведенным выше правилам, где вместо соответствующего операнда используется массив, имеющий этот операнд в качестве своего единственного компонента, и с нижней границей, совпадающей с нижней границей подтипа индекса.

Операцией катенации возбуждается исключение CONSTRAINT_ERROR, если верхняя граница результата выходит за диапазон подтипа индекса, кроме того случая, когда результат – пустой массив, а также если какой-либо операнд, тип которого есть тип компонента, имеет значение, не принадлежащее данному подтипу компонента.

Примеры:

```
X + 0.1          -- X должен иметь вещественный тип
"A" & "БВГ"      -- катенация двух строковых литералов
'A' & "БВГ"      -- катенация символьного и строкового литералов
'A' & 'A'        -- катенация двух символьных литералов
```

4.5.4. Унарные аддитивные операции

Унарные аддитивные операции + и – предопределены для любого числового типа и имеют свой обычный смысл. Для этих операций операнд и результат имеют один и тот же тип (см. табл. 4.5).

Таблица 4.5

Знак операции	*Операция*	*Тип операнда*	*Тип результата*
+	Сохранение знака	Любой числовой тип	Тот же числовой тип
–	Изменение знака	Любой числовой тип	Тот же числовой тип

4.5.5. Мультипликативные операции

Операции * и / предопределены для любого целого и любого плавающего типа и имеют свой обычный смысл. Операции **mod** и **rem** предопределены для любого целого типа. Для каждой из этих операций операнды и результат имеют один и тот же базовый тип. Для плавающих типов точность результата определяется типом операндов (см. 4.5.7). Характеристики операций сведены в табл. 4.6.

Таблица 4.6

Знак операции	*Операция*	*Тип операнда*	*Тип результата*
*	Умножение	Любой целый тип	Тот же целый тип
		Любой плавающий тип	Тот же плавающий тип
/	Целое деление	Любой целый тип	Тот же целый тип
	Деление плавающих	Любой плавающий тип	Тот же плавающий тип
mod	Вычет по модулю	Любой целый тип	Тот же целый тип
rem	Остаток	Любой целый тип	Тот же целый тип

Целое деление и остаток связаны следующим соотношением:

A = (A/B) * B + (A **rem** B),

где (A **rem** B) имеет знак значения A и абсолютное значение, меньшее абсолютного значения B. Целое деление удовлетворяет следующему тождеству:

(–A)/B = –(A/B) = A/(–B)

Результат операции вычета по модулю таков, что (A **mod** B) имеет знак значения B и абсолютное значение, меньшее абсолютного значения B, и существует целое значение K такое, что должно удовлетворяться следующее соотношение:

A = B * K + (A **mod** B)

Для каждого фиксированного типа предопределены операции умножения и деления на операнд предопределенного типа INTEGER (см. табл. 4.7).

Таблица 4.7

Знак операции	*Операция*	*Тип левого операнда*	*Тип правого операнда*	*Тип результата*
*	Умножение	Любой фиксированный тип	INTEGER	Тип левого операнда
		INTEGER	Любой фиксированный тип	Тип правого операнда
/	Деление	Любой фиксированный тип	INTEGER	Тип левого операнда

Умножение значения фиксированного типа на целое эквивалентно повторению операции сложения. Деление значения фиксированного типа на целое является приближенным и не меняет типа (см. 4.5.7).

Две специальные операции умножения и деления, применимые к операндам любых фиксированных типов, описаны в предопределенном пакете STANDARD (вследствие других правил они не могут быть переименованы или даны в качестве фактических параметров настройки). Соответствующие характеристики приведены в табл. 4.8.

Таблица 4.8

Знак операции	*Операция*	*Тип левого операнда*	*Тип правого операнда*	*Тип результата*
*	Умножение	Любой фиксированный тип	Любой фиксированный тип	*универсальный_фиксированный тип*
/	Деление	Любой фиксированный тип	Любой фиксированный тип	*универсальный_фиксированный тип*

Операнды умножения могут быть одного и того же или различных фиксированных типов, а тип результата – анонимный предопределенный *универсальный_фиксированный*, дельта которого произвольно мала. Результат любого такого умножения всегда должен быть явно преобразован в значение некоторого числового типа. Это обеспечивает явное управление точностью вычислений. То же относится к делению значения фиксированного типа на другое значение фиксированного типа. Никакие другие операции для типа *универсальный_фиксированный* не определены.

Исключение NUMERIC_ERROR возбуждается операциями целого деления, **rem** и **mod**, если правый операнд равен нулю.

Примеры:

```
I: INTEGER: = 1;
J: INTEGER: = 2;
K: INTEGER: = 3;
X: ВЕЩЕСТВ digits 6: = 1.0; - - см. 3.5.7
Y: ВЕЩЕСТВ digits 6: = 2.0;
F: ДРОБЬ delta 0.0001: = 0.1; - - см. 3.5.9
G: ДРОБЬ delta 0.0001: = 0.1;
```

Характеристики результатов операций над объектами из примеров приведены в табл. 4.9.

Таблица 4.9

Выражение	*Значение*	*Тип результата*
I * J	2	Тот же, что тип I и J, т. е. INTEGER
K/J	1	Тот же, что тип K и J, т. е. INTEGER
K mod J	1	Тот же, что тип K и J, т. е. INTEGER
X/Y	0.5	Тот же, что тип X и Y, т. е. ВЕЩЕСТВ
F/2	0.05	Тот же, что тип F, т. е. ДРОБЬ
3 * F	0.3	Тот же, что тип F, т. е. ДРОБЬ
F * G	0.01	*универсальный_фиксированный* требуется преобразование
ДРОБЬ (F * G)	0.01	ДРОБЬ в результате преобразования
ВЕЩЕСТВ (J) * Y	4.0	ВЕЩЕСТВ, как и тип обоих операндов после преобразования

Примечание. Для положительных A и B A/B – задает частное, а A rem B – остаток от деления A на B. Операция rem удовлетворяет следующим соотношениям:

A rem (–B) = A rem B

(–A) rem B = – (A rem B)

Для любого целого K справедливо следующее тождество:

A mod B = (A + K * B) mod B

Соотношения между целым делением, остатком и вычетом по модулю иллюстрируется табл. 4.10.

Таблица 4.10

A	B	A/B	A rem B	A mod B	A	B	A/B	A rem B	A mod B
10	5	2	0	0	–10	5	–2	0	0
11	5	2	1	1	–11	5	–2	–1	4
12	5	2	2	2	–12	5	–2	–2	3
13	5	2	3	3	–13	5	–2	–3	2
14	5	2	4	4	–14	5	–2	–4	1
10	–5	–2	0	0	–10	–5	2	0	0
11	–5	–2	1	–4	–11	–5	2	–1	–1
12	–5	–2	2	–3	–12	–5	2	–2	–2
13	–5	–2	3	–2	–13	–5	2	–3	–3
14	–5	–2	4	–1	–14	–5	2	–4	–4

4.5.6. Операции высшего приоритета

Унарная операция высшего приоритета **abs** предопределена для любого числового типа. Унарная операция высшего приоритета **not** предопределена для любого логического типа и любого одномерного индексируемого типа с компонентами логического типа. Характеристики операций приведены в табл. 4.11.

Таблица 4.11

Знак операции	*Операция*	*Тип операнда*	*Тип результата*
abs	Абсолютное значение	Любой числовой тип	Тот же числовой тип
not	Логическое отрицание	Любой логический тип Одномерный массив с логическими компонентами	Тот же логический тип Тот же индексируемый тип

Операция **not**, применяемая к одномерному массиву с логическими компонентами, вырабатывает одномерный логический массив с теми же самыми границами; каждый компонент результата получается как логическое отрицание соответствующего компонента операнда (т. е. компонента с тем же значением индекса).

Операция возведения в степень ** предопределена для каждого целого и для каждого плавающего типов (табл. 4.12). В обоих случаях правый операнд, называемый показателем степени, имеет предопределенный тип INTEGER.

Таблица 4.12

Знак операции	*Операция*	*Тип левого операнда*	*Тип правого операнда*	*Тип результата*
* *	Возведение в степень	Любой целый тип Любой плавающий тип	INTEGER INTEGER	Тип левого операнда Тип левого операнда

Возведение в степень с положительным показателем эквивалентно повторяющемуся умножению операнда на себя слева направо, в соответствии со значением показателя. Для операнда плавающего типа показатель степени может быть отрицательным, в этом случае результат – обратная величина результата с положительной степенью. Возведение в нулевую степень дает в результате единицу. Возведение в степень значения плавающего типа является приближенным (см. 4.5.7). При возведении целого значения в отрицательную степень возбуждается исключение CONSTRAINT_ERROR.

4.5.7. Точность операций с вещественными операндами

Вещественный подтип определяет множество модельных чисел. В терминах модельных чисел определяются точность, с которой базовые и предопределенные операции вырабатывают вещественный результат, и результат предопределенных операций отношения с вещественными операндами.

Модельный интервал подтипа – это интервал с границами, являющимися модельными числами этого подтипа. Связанный с принадлежащими вещественному подтипу значением, модельный интервал является наименьшим модельным интервалом (данного подтипа), который содержит это зна-

чение (модельный интервал, связанный с модельным числом некоторого подтипа, состоит только из этого числа).

Для любой базовой или предопределенной операции, вырабатывающей результат вещественного подтипа, требуемые границы результата задаются модельным интервалом, который определяется следующим образом:

• Модельный интервал результата – это наименьший модельный интервал (подтипа результата), который включает в себя минимальное и максимальное из всех значений, получаемых при применении (точной) математической операции, где каждый операнд – это любое значение из модельного интервала (подтипа операнда), определенного для этого операнда.

• Модельный интервал операнда, который сам является результатом операции, отличной от неявного преобразования, является модельным интервалом результата этой операции.

• Модельный интервал операнда, значение которого получено неявным преобразованием универсального выражения, является модельным интервалом, соответствующим этому значению из подтипа операнда.

Модельный интервал результата неопределен, если абсолютное значение хотя бы одного из упомянутых выше математических результатов превышает наибольшее хранимое число типа результата. Всякий раз, когда модельный интервал результата неопределен, и реализация не может обеспечить, чтобы фактический результат лежал в диапазоне хранимых чисел, крайне желательно возбуждение исключения NUMERIC_ERROR. Правилами языка, однако, этого не требуется, так как для некоторых объектных машин нет простых методов обнаружения переполнения. Значение атрибута MACHINE_OVERFLOWS указывает, возбуждает ли объектная машина исключение NUMERIC_ERROR в ситуациях переполнения (см. 13.7.3).

Хранимые числа вещественного типа определены (см. 3.5.6) как надмножество модельных чисел, границы ошибок которых подчиняются тем же правилам, что и для модельных чисел. Любое задаваемое в этом разделе в терминах модельных интервалов определение может поэтому быть распространено на хранимые интервалы хранимых чисел. Следствием такого распространения является то, что для реализации не допускается возбуждение исключения NUMERIC_ERROR, если интервал результата является хранимым интервалом.

Для результата операции возведения в степень модельный интервал, определяющий границы результата, определяется по приведенным выше правилам, которые применяются к последовательным умножениям при вычислении степени, и к заключительному делению, если показатель степени – отрицательное число.

Для результата операции отношения между двумя вещественными операндами рассмотрим модельный интервал (подтипа операнда), определенный для каждого такого операнда; результат может быть любым значением, полученным при применении математического сравнения к значениям, произвольно выбранным в соответствующих модельных интервалах операндов. Если один или оба модельных интервала операндов не определены

(и если при вычислении операндов не было возбуждено исключение), то в качестве результата сравнения допустимо любое возможное значение (т. е. либо TRUE, либо FALSE).

Результат проверки принадлежности определен в терминах сравнения значения операнда с нижней и верхней границами заданного диапазона или обозначения типа (к таким сравнениям применяются обычные правила).

Примечание. Для плавающего типа числа 15.0, 3.0 и 5.0 всегда являются модельными числами. Следовательно, X/Y, где X равно 15.0, а Y равно 3.0, согласно приведенным выше правилам вырабатывает в результате точно 5.0. В общем случае деление не вырабатывает в результате модельные числа и, следовательно, нельзя рассчитывать, что выполнено равенство (1.0/X) * X = 1.0.

4.6. **Преобразование типа**

Вычисление явного преобразования типа — это вычисление выражения, заданного в качестве операнда, и преобразование результата в значение указанного целевого типа. Явные преобразования типов допустимы между взаимосвязанными типами.

преобразование_типа : : = обозначение_типа (выражение)

Целевой тип преобразования типа — это базовый тип обозначения типа. Тип операнда преобразования типа должен быть определимым независимо от контекста (в частности, независимо от целевого типа). Более того, недопустимо, чтобы операнд преобразования типа был литералом **null**, генератором, агрегатом или строковым литералом; задание в качестве операнда преобразования типа выражения, заключенного в скобки, допускается только если само выражение является допустимым.

Преобразование к подтипу состоит в преобразовании к целевому типу с последующей проверкой принадлежности результата этому подтипу. Допускается преобразование операнда заданного типа к тому же самому типу.

Другие явные преобразования типов допустимы в следующих трех случаях.

а) Числовые типы.

Значение операнда любого числового типа может быть преобразовано в значение целевого типа, который должен быть также числовым. Для преобразования вещественных типов точность результата лежит в пределах точности заданного подтипа (см. 4.5.7). Преобразование значения вещественного типа в значение целого типа состоит в его округлении до ближайшего целого; для вещественного операнда, равноотстоящего от двух целых (с точностью этого вещественного подтипа), округление может быть произведено как в ту, так и в другую сторону.

б) Производные типы.

Преобразование допустимо, если либо целевой тип, либо тип операнда являются производными один от другого, непосредственно или косвенно, или если существует третий тип, от которого производными являются оба и тип операнда, и целевой тип, непосредственно или косвенно.

в) Индексируемые типы.

Преобразование допустимо, если тип операнда и целевой тип — индексируемые типы, которые удовлетворяют следующим условиям: оба типа

должны иметь одну и ту же размерность; в каждой позиции индекса типы индексов должны быть либо одинаковыми, либо взаимопреобразуемыми; типы компонентов должны быть одинаковыми; наконец, если тип компонента – тип с дискриминантами или ссылочный тип, то подтипы компонентов должны быть оба либо ограниченными, либо неограниченными.

Если обозначение типа задает неограниченный индексируемый тип, то для каждой позиции индекса границы результата преобразования определены преобразованием границ операнда в значения соответствующего типа индекса целевого типа. Если обозначение типа задает ограниченный индексируемый тип, то границы результата совпадают с границами, указанными в обозначении типа. В обоих случаях значение каждого компонента результата определяется соответствующим компонентом операнда (см. 4.5.2).

При вычислении преобразований числовых и производных типов возбуждается исключение CONSTRAINT_ERROR, если результат преобразования не удовлетворяет заданным в обозначении типа ограничениям.

При преобразовании индексируемых типов проверяется, что любое ограничение подтипа компонента одинаково для индексируемого типа операнда и для целевого индексируемого типа. Если обозначение типа задает неограниченный индексируемый тип и если операнд не является пустым массивом, то для каждой позиции индекса проверяется принадлежность границ результата соответствующему подтипу индекса целевого типа. Если обозначение типа задает ограниченный индексируемый подтип, то для каждого компонента операнда проверяется наличие соответствующего компонента целевого подтипа, и наоборот. Если хотя бы одна из этих проверок дает отрицательный результат, возбуждается исключение CONSTRAINT_ERROR.

Если допустимо преобразование одного типа в другой, то также допустимым является и обратное преобразование. Это обратное преобразование используется тогда, когда фактический параметр вида **in out** или **out** имеет форму преобразования типа имени (переменной), как это поясняется в разд. 6.4.1.

Единственной допустимой формой неявного преобразования типа является преобразование значения *универсального_целого* или *универсального_вещественного* типов в значение другого числового типа. Неявное преобразование операнда *универсального_целого* типа в другой целый тип или операнда *универсального_вещественного* типа в другой вещественный тип возможно только для операнда, являющегося либо числовым литералом, либо именованным числом, либо атрибутом; в этом разделе такой операнд называется *преобразуемым* универсальным операндом. Неявное преобразование преобразуемого универсального операнда применимо тогда и только тогда, когда самый вложенный полный контекст (см. 8.7) определяет единственный (числовой) целевой тип для этого неявного преобразования, и, кроме этого преобразования, не существует иной, предписанной языком, правильной интерпретации этого контекста.

Примечание. Правила для неявных преобразований подразумевают, что для операнда явного преобразования типа не производится никаких неявных преобразований.

Аналогично, не производится никаких неявных преобразований для операндов предопределенных операций отношения, являющихся преобразуемыми универсальными операндами.

Для индексируемых типов в языке допускается неявное преобразование подтипов (см. 5.2.1). Последствием явного преобразования типа может быть изменение представления (в частности, см. 13.6). Явные преобразования используются также для фактических параметров (см. 6.4).

Примеры преобразования числовых типов:

```
ВЕЩЕСТВ (2 * К) - - значение преобразуется к плавающему типу
INTEGER (1.6) - - значение равно 2
INTEGER (–0.4) - - значение равно 0
```

Примеры преобразования производных типов:

```
type A_ФОРМ is new B_ФОРМ;
X:A_ФОРМ;
Y: B_ФОРМ;
X: = A_ФОРМ (Y);
Y: = B_ФОРМ (X); - - обратное преобразование
```

Примеры преобразований индексируемых типов:

```
type ПОСЛЕДОВАТЕЛЬНОСТЬ is array (INTEGER range < >) of INTEGER;
subtype ДЮЖИНА is ПОСЛЕДОВАТЕЛЬНОСТЬ (1..12);
ЖУРНАЛ: array (1..100) of INTEGER;
ПОСЛЕДОВАТЕЛЬНОСТЬ (ЖУРНАЛ) - - с границами как у ЖУРНАЛ
ПОСЛЕДОВАТЕЛЬНОСТЬ (ЖУРНАЛ (31..42)) - - с границами 31–42
ДЮЖИНА (ЖУРНАЛ (31..42)) - - с границами, как у ДЮЖИНА
```

Примеры неявных преобразований:

```
X: INTEGER : = 2;
X + 1 + 2 - - неявное преобразование каждого целого литерала
1 + 2 + X - - неявное преобразование каждого целого литерала
X + (1 + 2) - - неявное преобразование каждого целого литерала
2 = (1 + 1) - - тип – универсальный_целый, неявных преобразований нет
A'LENGTH = B'LENGTH - - то же, что и выше
C: constant : = 3 + 2; - - то же, что и выше
X = 3 and 1 = 2 - - неявное преобразование 3, но не 1 и 2
```

4.7. Квалифицированные выражения

Квалифицированное выражение используется для явного указания типа и, возможно, подтипа операнда, заданного выражением или агрегатом.

квалифицированное_выражение : : =

обозначение_типа' (выражение) | обозначение_типа агрегат

Типом операнда должен быть базовый тип обозначения типа. Значение квалифицированного выражения – это значение операнда. Вычисление квалифицированного выражения выдает операнд и проверяет, принадлежит ли его значение подтипу, заданному в обозначении типа. При отрицательном результате проверки возбуждается исключение CONSTRAINT_ERROR.

Примеры:

```
type МАСКА is (FIX, DEC, EXP, SIGNIF);
type КОП is (FIX, CLA. DEC, TNZ, SUB);
PRINT (МАСКА' (DEC)); - - DEC типа МАСКА
PRINT (КОП' (DEC)); - - DEC типа КОП
for K in КОП' (FIX) .. КОП' (DEC) loop ...
- - квалификация необходима либо для FIX, либо для DEC
```

```
for K in КОП range FIX. . DEC loop . . .
- - квалификация не нужна
for K in КОП' (FIX) . . DEC loop . . .
- - квалификация для DEC не нужна
ДЮЖИНА' (1|3|5|7 => 2, others => 0) - - см. 4.6.
```

Примечание. Когда тип литерала перечисления или агрегата неизвестен из контекста, квалифицированное выражение может быть использовано для явного установления типа. Например, совмещенный литерал перечисления должен быть квалифицирован в следующих случаях: при использовании его в качестве параметра в вызове совмещенной подпрограммы, которая не может быть идентифицирована на основе типов остальных параметров и типа результата; в отношении, в котором оба операнда – совмещенные литералы перечисления; в массиве или диапазоне параметра цикла, в которых обе границы – совмещенные литералы перечисления. Явная квалификация используется также для определения совмещенной функции без параметров или для ограничения значения данным подтипом.

4.8. **Генераторы**

Вычисление генератора создает объект и вырабатывает ссылочное значение, которое указывает этот объект.

генератор : : = new указание_подтипа
| new квалифицированное_выражение

Тип созданного генератором объекта – это базовый тип обозначения типа, заданного либо в указании подтипа, либо в квалифицированном выражении. Для генератора с квалифицированным выражением это выражение задает начальное значение создаваемого объекта. Тип ссылочного значения, вырабатываемого генератором, должен быть определимым только из контекста, с учетом того, что это значение является ссылкой на указанный в генераторе тип.

Ограничение индекса и ограничение дискриминанта являются единственными допустимыми формами ограничения в указании подтипа генератора. Если в генераторе стоит указание подтипа, и если порождаемый объект имеет индексируемый тип или тип с дискриминантами, которые не содержат выражений по умолчанию, то указание подтипа должно либо обозначать ограниченный подтип, либо содержать явное ограничение индекса или дискриминанта.

Создаваемый объект индексируемого типа или типа с дискриминантами всегда ограничен. Для генератора с указанием подтипа создаваемый объект ограничен либо этим подтипом, либо значениями дискриминанта по умолчанию. Для генератора с квалифицированным выражением создаваемый объект ограничен границами или дискриминантами начального значения. Для других типов создаваемый объект имеет подтип, определенный указанием подтипа в определении ссылочного типа.

При вычислении генератора сначала производится предвыполнение указания подтипа или вычисление квалифицированного выражения. Затем создается новый объект. Далее осуществляется инициализация как для описанного объекта (см. 3.2.1): явно – для квалифицированного выражения, неявно – для указания подтипа. Наконец, возвращается ссылочное значение, указывающее на созданный объект.

Реализация должна гарантировать сохранение объекта, созданного при вычислении генератора, до тех пор, пока сам объект или хотя бы один из его подкомпонентов доступны непосредственно или косвенно, т. е. пока он может быть обозначен некоторым именем. Кроме того, если объект или один из его подкомпонентов принадлежат задачному типу, он считается доступным, пока не завершена эта задача. Реализация может (но не обязана) освобождать память, занятую объектом, созданным генератором, как только этот объект становится недоступным.

В случаях, когда требуется более точное управление распределением памяти под указанные значениями ссылочного типа объекты, оно может быть обеспечено следующими средствами.

а) Общий объем памяти, доступный для набора объектов ссылочного типа, может быть установлен с помощью спецификатора длины (см. 13.2).

б) Прагма CONTROLLED сообщает реализации, что для объектов, указанных значениями ссылочного типа, автоматическое возвращение памяти производиться не должно, исключая случаи выхода из самого вложенного оператора блока, тела подпрограммы или тела задачи, содержащих описание этого ссылочного типа, или выхода из главной программы.

pragma CONTROLLED (простое_имя_*ссылочного_типа*);

Эта прагма для данного ссылочного типа допустима в тех же местах, что и спецификатор представления этого типа (см. 13.1).

в) Явного освобождения памяти, занимаемой объектом, указанным ссылочным значением, можно достичь вызовом процедуры, полученной настройкой предопределенной библиотечной процедуры UNCHECKED_DEALLOCATION (см. 13.10.1).

При исчерпании памяти генератором возбуждается исключение STORAGE_ERROR. Заметим также, что исключение CONSTRAINT_ERROR может быть возбуждено при вычислении квалифицированного выражения, во время предвыполнения указания подтипа или при инициализации.

Примеры (ссылочных типов, описанных в разд. 3.8):

```
new ЯЧЕЙКА'(0, null, null)                                  - - явная инициализация
new ЯЧЕЙКА'(ЗНАЧЕНИЕ => 0, СЛЕД => null, ПРЕД => null)
                                                            - - явная инициализация
new ЯЧЕЙКА                                                  - - нет инициализации
new МАТРИЦА'(1..10, 1..20)                                  - - даны только границы
new МАТРИЦА'(1..10 => (1..20 => 0.01))                      - - явная инициализация
new БУФЕР(100)                                              - - дан только дискриминант
new БУФЕР'(РАЗМЕР => 80, ПОЗ => 0, ЗНАЧЕНИЕ => (1..80 => 'A'))
                                                            - - явная инициализация
```

4.9. **Статические выражения и статические подтипы**

Некоторые выражения скалярного типа называются *статическими.* Аналогично, статическими называют некоторые дискретные диапазоны, а обозначения типов для некоторых скалярных подтипов называют обозначающими статические подтипы.

Выражение скалярного типа называется статическим тогда и только тогда, когда каждое первичное является одним из перечисленных в под-

пунктах от *а* до *з*, а каждая операция – это предопределенная операция и вычисление выражения дает значение (т. е. не возбуждает исключения):

а) литерал перечисления (включая символьный литерал);

б) числовой литерал;

в) именованное число;

г) заданная явным описанием константа статического подтипа, инициализованная статическим выражением;

д) вызов функции, имя которой – знак операции, обозначающий предопределенную операцию, включая расширенное имя; каждый фактический параметр должен быть статическим выражением;

е) определяемый в языке атрибут статического подтипа; если атрибут – функция, фактические параметры должны быть также статическими выражениями;

ж) квалифицированное выражение, обозначение типа которого задает статический подтип, а операнд – статическое выражение;

з) заключенное в скобки статическое выражение.

Статическим является диапазон, границы которого – статические выражения. Статическим является ограничение диапазона, если составляющие его атрибут диапазона или простые выражения являются статическими. Статический подтип – это либо скалярный базовый тип, отличный от формального типа настройки, либо скалярный подтип, образованный наложением на статический подтип, либо ограничения статическим диапазоном, либо ограничения плавающего или фиксированного типа, ограничение диапазона которого, если оно есть, является статическим. Статический дискретный диапазон – это либо статический подтип, либо статический диапазон. Статическое ограничение индекса – это ограничение индекса, для которого статическим является каждый подтип индекса соответствующего индексируемого типа и для которого статическим является каждый дискретный диапазон. Статическое ограничение дискриминанта –это ограничение дискриминанта, для которого статическим является подтип каждого дискриминанта и в котором статическим является каждое выражение.

Примечание. Точность вычисления статического выражения вещественного типа определена правилами в разд. 4.5.7. Если результат не является модельным (или хранимым) числом этого типа, то значение выражения, полученное при вычислении во время компиляции, не обязано совпадать со значением, которое получится при вычислении во время счета.

Атрибуты массивов не являются статическими, в частности, статическим не является атрибут RANGE.

4.10. **Универсальные выражения**

Универсальное_выражение – это выражение, вырабатывающее результат *универсального_целого* или *универсального_вещественного* типа.

Для *универсального_целого* типа предопределены те же операции, что и для любого целого типа. Для *универсального_вещественного* типа предопределены те же операции, что и для любого плавающего типа. Кроме того, эти операции включают операции умножения и деления (табл. 4.13).

Таблица 4.13

Знак операции	*Операция*	*Тип левого операнда*	*Тип правого операнда*	*Тип результата*
*	Умножение	Универсальный_вещественный	Универсальный_целый	Универсальный_вещественный
		Универсальный_целый	Универсальный_вещественный	Универсальный_вещественный
/	Деление	Универсальный_вещественный	Универсальный_целый	Универсальный_вещественный

Точность вычисления универсального выражения типа *универсальный_вещественный* обязана быть не ниже точности любого из предопределенных плавающих типов, поддержанных в реализации, исключая сам *универсальный_вещественный* тип. Более того, если универсальное выражение – статическое, то вычисление должно быть точным.

При вычислении операций универсального выражения, не являющегося статическим, возбуждение исключения NUMERIC_ERROR реализацией допускается только в том случае, если результат операции – вещественное число с абсолютным значением, превышающим наибольшее хранимое число самого точного предопределенного плавающего типа (исключая *универсальный_вещественный*), или целое значение большее, чем SYSTEM.MAX_INT, либо меньшее, чем SYSTEM.MIN_INT.

Примечание. Следствием приведенных выше правил является то, что тип универсального выражения – *универсальный_целый*, если этот тип имеет каждое первичное, содержащееся в выражении (исключая фактические параметры атрибутов – функций и правые операнды операций возведения в степень), в противном случае тип универсального выражения – *универсальный_вещественный*.

Примеры:

```
1 + 1 -- 2
abs (–10) * 3 -- 30
КИЛО: constant : = 1000;
МЕГА: constant : = КИЛО * КИЛО; -- 1_000_000
ДЛИНА: constant : = FLOAT'DIGITS * 2;
ПОЛ_ПИ: constant : = ПИ/2; -- см. 3.2.2
ГРАД_В_РАД: constant : = ПОЛ_ПИ/90;
РАД_В_ГРАД: constant : = 1.0/ГРАД_В_РАД;
-- эквивалентно 1.0/((3.14159_26536/2)/90)
```

5. ОПЕРАТОРЫ

Оператор определяет выполняемое действие; процесс осуществления этого действия называется *выполнением* оператора.

В данной главе описываются общие правила, применимые ко всем операторам, и некоторые операторы. Оператор вызова процедуры описывается в гл. 6. Операторы вызова входа, задержки, принятия, отбора и прекращения описываются в гл. 9, оператор возбуждения – в гл. 11, а оператор кода – в гл. 13.

5.1. **Простые и составные операторы. Последовательности операторов**

Оператор может быть или простым или составным. Простой оператор не содержит других операторов. Составной оператор содержит простые операторы и другие составные операторы.

последовательность_операторов : : = оператор {оператор}

оператор : : = {метка} простой_оператор | {метка}составной_оператор

простой_оператор : : = пустой_оператор

| оператор_присваивания | оператор_вызова_процедуры
| оператор_выхода | оператор_возврата
| оператор_перехода | оператор_вызова_входа
| оператор_задержки | оператор_прекращения
| оператор_возбуждения | оператор_кода

составной_оператор : : =

условный_оператор | оператор_выбора
| оператор_цикла | оператор_блока
| оператор_принятия | оператор_отбора

метка : : = <<простое_имя_*метки*>>

пустой_оператор : : = **null**;

Говорят, что имя каждой метки, стоящей перед оператором, *помечает* этот оператор. Имя метки (а также имя цикла или блока) неявно описано в конце раздела описаний самого внутреннего блока, тела подпрограммы, тела пакета, тела задачи или настраиваемого тела, которые содержат данный помеченный оператор (именованный оператор цикла или именованный оператор блока). При отсутствии в операторе блока раздела описаний подразумевается, что оператор блока содержит неявный раздел описаний (с предшествующим **declare**).

Имена меток, циклов и блоков неявно описываются в порядке появления в тексте программы начал соответствующих помеченных операторов, операторов цикла и операторов блока. Для имен меток, циклов и блоков, неявно описанных в теле программного модуля, включая вложенные в это тело операторы блока, но исключая другие вложенные программные модули (т. е. подпрограммы, пакеты, задачи или настраиваемые модули), должны использоваться различные идентификаторы.

Выполнение пустого оператора заключается в переходе к выполнению следующего оператора.

Выполнение последовательности операторов состоит в поочередном выполнении отдельных операторов последовательности, пока они все не будут закончены или пока не произойдет передача управления. Передача управления вызывается выполнением операторов выхода, возврата или перехода, выбором альтернативы завершения, возбуждением исключения или (неявно) выполнением оператора прекращения.

Примеры помеченных операторов:

```
<<ЗДЕСЬ>><<КТО>><<ПУСТ>><<ТАМ>> null;
<<ПОСЛЕ>> X: = 1;
```

Примечание. Область действия описания начинается самим описанием (см. 8.2). Область действия *неявного* описания имени метки, цикла или блока начинается до первого *явного* вхождения соответствующего имени, поскольку это вхождение возможно лишь в качестве метки оператора, в операторе блока, операторе цикла или операторе перехода. Неявное описание в операторе блока может скрыть описание, данное во внешнем программном модуле или операторе блока (согласно обычным правилам скрытия, изложенным в разд. 8.3).

5.2. **Операторы присваивания**

Оператор присваивания заменяет текущее значение переменной новым значением, задаваемым выражением. Переменная в левой части оператора присваивания и выражение в его правой части должны быть одного и того же типа, однако тип не должен быть лимитируемым.

оператор_присваивания : : = имя_*переменной* : = выражение;

При выполнении оператора присваивания вначале вычисляются имя переменной и выражение в некотором порядке, не определенном в языке. Затем, если переменная не является массивом, проверяется принадлежность значения выражения подтипу переменной (если же переменная – массив, то при присваивании производится преобразование подтипа, как описано в разд. 5.2.1). Наконец, значение выражения становится новым значением переменной.

При отрицательном результате упомянутой выше проверки принадлежности подтипу возбуждается исключение CONSTRAINT_ERROR, а текущее значение переменной не изменяется. Если переменная является подкомпонентом, зависящим от дискриминантов неограниченной именуемой переменной, то выполнение присваивания ошибочно, если при этом выполнении изменяется значение любого из этих дискриминантов.

Примеры:

```
ЗНАЧЕНИЕ : = МАКС_ЗНАЧЕНИЕ – 1;
ОТТЕНОК : = ГОЛУБОЙ;
СЛЕДУЮЩЕЕ_ОБРАМЛЕНИЕ (F) (M, N) : = 2.5 - - см. 4.1.1
A: = СКАЛ_ПРОИЗВЕДЕНИЕ (B, C);                - - см. 6.5
ПИШУЩЕЕ_УСТРОЙСТВО : = (СОСТОЯНИЕ => ОТКРЫТО, УСТРОЙСТВО =>
          ПЕЧАТЬ, КОЛ_СТРОЧЕК => 60); - - см. 3.7.3
СЛЕДУЮЩИЙ_АВТО. all : = (72074, null); - - см. 3.8.1
```

Примеры проверок ограничений:

```
I, J: INTEGER range 1. .10;
K: INTEGER range 1. .20;
I: = J; - - одинаковые диапазоны
K: = J; - - совместимые диапазоны
J: = K; - - при K > 10 возбудится исключение CONSTRAINT_ERROR
```

Примечание. Значения дискриминантов объекта, указанного ссылочным значением, не могут быть изменены (даже присваиванием составного значения всему объекту), поскольку такие объекты, созданные генераторами, всегда являются ограниченными (см. 4.8); однако подкомпоненты таких объектов могут и не быть ограниченными.

Если выражение в правой части является числовым литералом, именованным числом или атрибутом, вырабатывающим результат типа *универсальный_целый* или *универсальный_вещественный,* то производится неявное преобразование типа, описанное в разд. 4.6.

Определение типа переменной в левой части оператора присваивания может потребовать рассмотрения выражения, если имя переменной может интерпретироваться как имя переменной, указанной ссылочным значением, возвращаемым вызовом функции, а также как компонент или отрезок такой переменной (см. разд. 8.7 о контексте разрешения совмещения).

5.2.1. Присваивания массивов

Если переменная в левой части оператора присваивания является индексируемой (в частности, отрезком), значение выражения неявно преобразуется в значение подтипа индексируемой переменной, после чего результат преобразования становится новым значением этой переменной.

Это значит, что новое значение каждого компонента индексируемой переменной задается сопоставляемым ей компонентом индексируемого значения, полученного при вычислении выражения (см. определение сопоставляемых компонентов в 4.5.2). При преобразовании подтипа для каждого компонента индексируемой переменной проверяется наличие сопоставляемого компонента индексируемого значения, и наоборот. При отрицательном результате этой проверки возбуждается исключение CONSTRAINT_ERROR, а значение каждого компонента переменной не изменяется.

Примеры:

```
A: STRING (1..31);
B: STRING (3..33);
A := B; -- одинаковое количество компонентов
A(1..9) := "ПРЕКРАСНО";
A(4..12) := A(1..9); -- A(1..12) = "ПРЕПРЕКРАСНО"
```

Примечание. Присваивание массива определено даже в случае перекрывающихся отрезков, поскольку выражение в правой части вычисляется до присваивания компонентов. Так, в случае последнего из приведенных примеров, реализация, вырабатывающая A(1..12) = "ПРЕПРЕПРЕПРЕ", была бы некорректной (покомпонентная передача недопустима).

Описанное выше явное преобразование подтипа выполняется только на уровне значения всего выражения правой части, но не для его подкомпонентов, являющихся индексируемыми значениями.

5.3. **Условные операторы**

Условный оператор выбирает для выполнения одну или ни одной из входящих в него последовательностей операторов, в зависимости от значения (истинности) одного или нескольких условий.

условный оператор : : =
if условие **then**
последовательность_операторов
{**elsif** условие **then**
последовательность_операторов}
[**else**
последовательность_операторов]
end if;

условие : : = *логическое*_выражение

Выражение, задающее условие, должно быть логического типа. Для выполнения условного оператора вычисляются последовательно условия после **if** и **elsif** (трактуя заключительное **else** как **elsif** TRUE **then**) до тех пор,

пока одно из них не скажется истинным или не будут исчерпаны все условия. При нахождении условия со значением TRUE выполняется соответствующая последовательность операторов, в противном случае не выполняется ни одна из последовательностей операторов.

Примеры:

```
if МЕСЯЦ = ДЕКАБРЬ and ДЕНЬ = 31 then
  МЕСЯЦ : = ЯНВАРЬ;
  ДЕНЬ : = 1;
  ГОД: = ГОД + 1;
end if;
if СТРОЧКА_СЛИШКОМ_КОРОТКА then
  raise LAYOUT_ERROR;
elsif СТРОЧКА_ЗАПОЛНЕНА then
  NEW_LINE,
  PUT (ЭЛЕМЕНТ);
else PUT (ЭЛЕМЕНТ);
end if;
if МОЙ_АВТО.ВЛАДЕЛЕЦ.МАШИНА/ = МОЙ_АВТО then
  СООБЩЕНИЕ ("НЕПРАВИЛЬНЫЕ ДАННЫЕ"); - - см. 3.8
end if;
```

5.4. **Операторы выбора**

Оператор выбора выбирает для выполнения одну из нескольких альтернативных последовательностей операторов; выбор альтернативы осуществляется в зависимости от значения выражения.

```
оператор_выбора : : =
  case выражение is
     альтернатива_оператора_выбора
     {альтернатива_оператора_выбора}
  end case;
альтернатива_оператора_выбора : : =
  when выбор {|выбор} =>
     последовательность_операторов
```

Выражение должно быть дискретного типа, который должен быть определимым независимо от контекста выражения, но с учетом того факта, что тип выражения должен быть дискретным. Более того, тип этого выражения не должен быть настраиваемым формальным типом. Каждый выбор в альтернативе оператора выбора должен быть того же типа, что и выражение; перечень выборов определяет, для каких значений выражения выбирается соответствующая альтернатива.

Если выражение является именем объекта статического подтипа, то каждое значение этого подтипа должно быть представлено один и только один раз в наборе выборов оператора выбора, и никакие другие значения недопустимы; это правило применяется также, если выражение является квалифицированным выражением или преобразованием типа, обозначение типа которого указывает статический подтип. В остальных случаях, для других форм выражения, каждое значение (базового) типа выражения должно быть представлено только один раз в наборе выборов, и никакие другие значения недопустимы.

Используемые в качестве выборов в операторе выбора простые выражения и дискретные диапазоны должны быть статическими. Выбор, являющийся дискретным диапазоном, задает все значения из этого диапазона (ни одного значения в случае пустого диапазона). Выбор **others** допустим только в качестве единственного выбора для последней альтернативы и задает все значения (возможно и ни одного), не заданных в выборах предыдущих альтернатив. В качестве выбора в альтернативе оператора выбора не допускается использование простого имени компонента.

Выполнение оператора выбора заключается в вычислении выражения, в выборе последовательности операторов и в выполнении выбранной последовательности операторов.

Примеры:

```
case СЕНСОР is
  when ВЫСОТА => ЗАПИСАТЬ_ВЫСОТУ (ПОКАЗАНИЕ_СЕНСОРА);
  when АЗИМУТ => ЗАПИСАТЬ_АЗИМУТ (ПОКАЗАНИЕ_СЕНСОРА);
  when РАССТОЯНИЕ => ЗАПИСАТЬ_РАССТОЯНИЕ (ПОКАЗАНИЕ_СЕНСОРА);
  when others => null;
end case;
case СЕГОДНЯ is
  when ПНД => ПОДСЧИТАТЬ_ ИСХОДНЫЙ_БАЛАНС;
  when ПТН => ПОДСЧИТАТЬ_ ИТОГОВЫЙ_БАЛАНС;
  when ВТР. .ЧТВ => ВЫДАТЬ_ОТЧЕТ (СЕГОДНЯ);
  when СББ. . ВСК => null;
end case;
case ДВ_ЧИСЛО (СЧЕТЧИК) is
  when 1 => ЗАМЕНИТЬ_ДВ (1);
  when 2 => ЗАМЕНИТЬ_ДВ (2);
  when 3|4 => ОБНУЛИТЬ_ДВ (1); ОБНУЛИТЬ_ДВ (1);
  when others => raise ОШИБКА;
end case;
```

Примечание. При выполнении оператора выбора выбирается одна и только одна альтернатива, так как выборы являются исчерпывающими и взаимно исключающими. Квалификацией выражения в операторе выбора статическим подтипом можно ограничить количество выборов, которые необходимо указать явно.

Выбор others обязателен в операторе выбора, если выражение имеет тип *универсальный_целый* (например, выражение является целым литералом), так как это единственный способ учесть все значения типа *универсальный_целый.*

5.5. Операторы цикла

Оператор цикла содержит последовательность операторов, выполнение которой повторяется несколько раз или ни одного раза.

```
оператор_цикла : : =
  [простое_имя_цикла :]
     [схема_итерации] loop
        последовательность_операторов
     end loop [простое_имя_цикла];
схема_итерации : : = while условие
  | for спецификация_параметра_цикла
спецификация_параметра_цикла : : =
  идентификатор in [reverse] дискретный_диапазон
```

Если в операторе цикла используется простое имя цикла, то оно должно задаваться как в начале, так и в конце этого оператора.

Оператор цикла без схемы итерации определяет повторяемое выполнение последовательности операторов. Выполнение такого оператора цикла заканчивается выходом из цикла вследствие выполнения оператора выхода или какой-либо другой передачи управления (см. 5.1).

Для оператора цикла со схемой итерации **while** перед каждым выполнением последовательности операторов вычисляется условие; если значением условия является TRUE, то последовательность операторов выполняется, если – FALSE, то выполнение оператора цикла заканчивается.

Для оператора цикла со схемой итерации **for** спецификация параметра цикла является описанием *параметра цикла* с заданным в схеме итерации идентификатором. Параметр цикла – это объект, типом которого является базовый тип значений дискретного диапазона (см. 3.6.1). В пределах последовательности операторов параметр цикла считается константой. Поэтому его использование в качестве переменной в левой части оператора присваивания недопустимо. Параметр цикла не должен использоваться в качестве параметра вида **out** или **in out** оператора вызова процедуры (или входа) или в качестве параметра вида **in out** конкретизации настройки.

Для выполнения оператора цикла со схемой итерации **for** сначала предвыполняется спецификация параметра цикла. При этом создается параметр цикла и вычисляется дискретный диапазон.

Если дискретный диапазон пуст, то выполнение оператора цикла заканчивается. В противном случае последовательность операторов выполняется по одному разу для каждого значения из дискретного диапазона (при условии, что не происходит выхода из цикла из-за выполнения оператора выхода или какой-либо другой передачи управления). Перед каждой такой итерацией параметру цикла присваивается соответствующее значение из дискретного диапазона. При отсутствии зарезервированного слова **reverse** эти значения присваиваются в порядке возрастания, при наличии этого слова – в порядке убывания.

Пример оператора цикла без схемы итерации:

```
loop
  GET (ТЕКУЩИЙ_СИМВОЛ);
  exit when ТЕКУЩИЙ_СИМВОЛ = '*';
end loop;
```

Пример оператора цикла со схемой итерации **while**:

```
while ЗАЯВКА (К). ЦЕНА < ПРЕДЕЛ. ЦЕНА loop
      ЗАПИСАТЬ_ЗАЯВКУ (ЗАЯВКА (К). ЦЕНА);
      К: = К + 1;
end loop;
```

Пример оператора цикла со схемой итерации **for**:

```
for J in БУФЕР' RANGE loop - - правильно даже для пустого диапазона
  if БУФЕР (J) / = ПРОБЕЛ then
     PUT (БУФЕР (J));
  end if;
end loop;
```

Пример оператора цикла с простым именем цикла:

```
СУММИРОВАНИЕ:
while СЛЕДУЮЩИЙ/ = ГОЛОВА loop -- см. 3.8.1
  СУМ: = СУМ + СЛЕДУЮЩИЙ. ЗНАЧЕНИЕ;
  СЛЕДУЮЩИЙ : = СЛЕДУЮЩИЙ. СЛЕД;
end loop СУММИРОВАНИЕ;
```

Примечание. Область действия параметра цикла простирается от спецификации параметра цикла до конца оператора цикла, а правила видимости таковы, что параметр цикла видим только в пределах последовательности операторов в цикле.

Дискретный диапазон цикла вычисляется только один раз. Использование зарезервированного слова reverse не изменяет дискретный диапазон, так что следующие схемы итерации не эквивалентны (в первой – диапазон пуст):

```
for J in reverse 1..0
for J in 0..1
```

Имена циклов используются также в операторах выхода и в расширенных именах (в качестве префикса имени параметра цикла).

5.6. **Операторы блока**

Оператор блока содержит последовательность операторов, которой может предшествовать раздел описаний и за которой могут следовать обработчики исключений.

```
оператор_блока : : =
  [простое_имя_блока :]
     [declare
        раздел_описаний]
     begin
        последовательность_операторов
     [exception
        обработчик_исключения
        {обработчик_исключения}]
     end [простое_имя_блока] ;
```

Если в операторе блока используется простое имя блока, то оно должно задаваться как в начале, так и в конце.

Выполнение оператора блока заключается в предвыполнении раздела описаний (при его наличии) и последующем выполнении последовательности операторов. Если оператор блока содержит обработчики исключений, то они выполняются при возбуждении соответствующих исключений во время выполнения последовательности операторов (см. 11.2).

Пример:

```
ОБМЕН:
  declare
     РАБОЧАЯ_ЯЧЕЙКА: INTEGER;
  begin
     РАБОЧАЯ_ЯЧЕЙКА: = A; A: = B; B: = РАБОЧАЯ_ЯЧЕЙКА;
  end ОБМЕН;
```

Примечание. Если в операторе блока, выполнение последовательности операторов которого окончено, описаны объекты задачного типа, то оператор блока не заканчивается до тех пор, пока не будут завершены все его подчиненные задачи (см. 9.4). Это правило применяется также при окончании из-за выполнения операторов выхода, возврата или перехода, или из-за возбуждения исключения.

Внутри оператора блока его имя может использоваться в расширенных именах локальных понятий, таких как ОБМЕН.РАБОЧАЯ_ЯЧЕЙКА в приведенном выше примере (см. 4.1.3*е*).

5.7. **Операторы выхода**

Оператор выхода используется для окончания выполнения объемлющего оператора цикла (называемого в дальнейшем просто циклом); окончание может быть условным, если оператор выхода содержит условие.

оператор_выхода : : = **exit** [простое_имя_*цикла*] [**when** условие] ;

Оператор выхода с именем цикла допустим только в именованном цикле и применяется к этому циклу; оператор выхода без имени цикла допускается в некотором цикле и применяется к самому внутреннему объемлющему циклу (именованному или нет). Кроме этого, применяемый к конкретному циклу оператор выхода не должен появляться в теле подпрограммы, теле пакета, теле задачи, в настраиваемом теле или в операторе принятия, если эта конструкция вложена в рассматриваемый цикл.

При выполнении оператора выхода сначала вычисляется условие, если оно есть. Выход из цикла происходит, если значением условия является TRUE или условие отсутствует.

Примеры:

```
for К in 1..МАКС_КОЛ_ЭЛЕМЕНТОВ loop
  ВВЕСТИ_НОВЫЙ_ЭЛЕМЕНТ (НОВЫЙ_ЭЛЕМЕНТ);
  ПРИСОЕДИНИТЬ_ЭЛЕМЕНТ (НОВЫЙ_ЭЛЕМЕНТ, ФАЙЛ_ПАМЯТИ);
  exit when НОВЫЙ_ЭЛЕМЕНТ = ЗАКЛЮЧИТЕЛЬНЫЙ_ЭЛЕМЕНТ;
end loop;
ГЛАВНЫЙ_ЦИКЛ:
loop;
  - - начальные операторы
  exit ГЛАВНЫЙ_ЦИКЛ when НАЙДЕНО;
     - - заключительные операторы
  end loop ГЛАВНЫЙ_ЦИКЛ:
```

Примечание. Выход из нескольких вложенных циклов можно осуществить с помощью оператора выхода с именем внешнего цикла.

5.8. **Операторы возврата**

Оператор возврата используется для окончания выполнения самой внутренней объемлющей конструкции, которая может быть функцией, процедурой или оператором принятия.

оператор_возврата : : = **return** [выражение] ;

Оператор возврата допустим только в теле подпрограммы (или настраиваемой подпрограммы) или в операторе принятия и применяется к самой внутренней из объемлющих его таких конструкций; оператор возврата недопустим в теле модуля-задачи, пакета или настраиваемого пакета, объемлемого одной из указанных конструкций (с другой стороны, он допустим в составном операторе, вложенном в такую конструкцию и, в частности, в операторе блока).

Оператор возврата в операторе принятия, в теле процедуры или настраиваемой процедуры не должен содержать выражения. Оператор возврата в теле функции или настраиваемой функции должен содержать выражение.

Значение выражения определяет результат, возвращаемый функцией. Тип этого выражения должен быть базовым для обозначения типа, приводимого после зарезервированного слова **return** в спецификации функции или настраиваемой функции (это обозначение типа определяет подтип результата).

При выполнении оператора возврата сначала вычисляется выражение (при его наличии) и проверяется принадлежность его значения подтипу результата. При положительном итоге проверки заканчивается вычисление оператора возврата и одновременно подпрограммы или оператора принятия, при отрицательном – в месте оператора возврата возбуждается исключение CONSTRAINT_ERROR.

Примеры:

```
return;                                                    - - в процедуре
return ЗНАЧЕНИЕ_КЛЮЧА (ПОСЛЕДНИЙ_ИНДЕКС); - - в функции
```

Примечание. Если выражение является числовым литералом, именованным числом или атрибутом, который вырабатывает результат типа *универсальный_целый* или *универсальный_вещественный*, то выполняется неявное преобразование результата, как описано в разд. 4.6.

5.9. Операторы перехода

Оператор перехода определяет явную передачу управления на помеченный меткой оператор.

оператор_перехода : : = **goto** имя_*метки;*

Самая вложенная последовательность операторов, объемлющая помеченный меткой оператор, должна также охватывать и оператор перехода на эту метку (в частности, оператор перехода может входить в еще более вложенную последовательность операторов). Кроме того, если оператор перехода содержится в операторе принятия или теле программного модуля, то соответствующий помеченный оператор не должен быть вне этой конструкции, и наоборот (как следует из предыдущего правила), если помеченный оператор содержится в такой конструкции, то оператор перехода не может быть вне ее.

Выполнение оператора перехода заключается в передаче управления на помеченный соответствующей меткой оператор.

Примечание. Приведенные выше правила допускают передачу управления на оператор из некоторой объемлющей последовательности операторов, но не наоборот. Аналогично, они запрещают передачу управления между альтернативами оператора выбора, условного оператора или оператора отбора, между обработчиками исключений или из обработчика исключения некоторого окружения назад на последовательность операторов этого окружения.

Пример:

```
<<СРАВНИТЬ>>
if A (I) < ЭЛЕМЕНТ then
  if ЛЕВЫЙ (I) / = 0 then
      I: = ЛЕВЫЙ (I);
      goto СРАВНИТЬ;
  end if;
  - - некоторые операторы
end if;
```

6. ПОДПРОГРАММЫ

Подпрограммы являются одной из четырех форм *программных модулей*, из которых могут быть составлены программы. Другие формы – это пакеты, задачные модули и настраиваемые модули.

Подпрограмма – это программный модуль, выполнение которого инициируется вызовом подпрограммы. Существуют две формы подпрограмм: процедуры и функции. Вызов процедуры – это оператор; вызов функции является выражением и возвращает значение. Определение подпрограммы может состоять из двух частей: описания подпрограммы, определяющего соглашения о ее вызове, и тела подпрограммы, определяющего ее выполнение.

6.1. **Описание подпрограммы**

Описание подпрограммы объявляет процедуру или функцию в зависимости от указанного начального зарезервированного слова.

описание_подпрограммы : : = спецификация_подпрограммы;
спецификация_подпрограммы : : =
procedure идентификатор [раздел_формальных_параметров]
| **function** обозначение [раздел_формальных_параметров]
return обозначение_типа
обозначение : : = идентификатор | знак_операции
знак_операции : : = строковый_литерал
раздел_формальных_параметров : : =
(спецификация_параметра {; спецификация_параметра})
спецификация_параметра : : =
список_идентификаторов : вид обозначение_типа [: = выражение]
вид : : = **[in]** | **in out** | **out**

Спецификация процедуры определяет ее идентификатор и ее *формальные параметры* (если они есть). Спецификация функции определяет ее обозначение, ее формальные параметры (если они есть) и подтип возвращаемого значения (*подтип результата*). Обозначение, являющееся знаком операции, используется для совмещения операций. Последовательность символов, представляющая знак операции, должна представлять операцию, принадлежащую одному из шести классов совмещаемых операций, определенных в разд. 4.5 (пробелы не допустимы, а для букв ограничений нет).

Спецификация параметра с несколькими идентификаторами эквивалентна последовательности спецификаций с одним параметром, как поясняется в разд 3.2. Каждая спецификация одного параметра описывает формальный параметр. Если вид явно не задан, то предполагается вид **in**. Если спецификация параметра оканчивается выражением, то оно является *выражением по умолчанию* формального параметра. Выражение по умолчанию допустимо только в спецификации параметра вида **in** (независимо от явного или неявного его указания). Тип выражения по умолчанию должен совпадать с типом соответствующего формального параметра.

Не допускается использование имени, обозначающего формальный параметр, в выражении по умолчанию, если спецификация этого параметра дана в том же разделе формальных параметров.

Предвыполнение описания подпрограммы предвыполняет соответствующий раздел формальных параметров. Предвыполнение раздела формальных параметров не дает другого эффекта.

Примеры описания подпрограмм:

```
procedure ОБХОД_ДЕРЕВА;
procedure УВЕЛИЧЕНИЕ (X: in out INTEGER);
procedure ПРАВЫЙ_АБЗАЦ (ПРЕДЕЛЫ: out РАЗМЕР_СТРОЧКИ); - - см. 3.5.4
procedure ПЕРЕКЛЮЧАТЕЛЬ (ИЗ, В: in out СВЯЗЬ); - - см. 3.8.1
function СЛУЧАЙНАЯ_ВЕЛИЧИНА return ВЕРОЯТНОСТЬ; - - см. 3.5.7
function МИН_ЯЧЕЙКА (X: СВЯЗЬ) return ЯЧЕЙКА; - - 3.8.1
function СЛЕДУЮЩЕЕ_ОБРАМЛЕНИЕ (К: POSITIVE) return ОБРАМЛЕНИЕ;
                                                    - - см. 3.8
function СКАЛ_ПРОИЗВЕДЕНИЕ (ЛЕВЫЙ, ПРАВЫЙ: ВЕКТОР) return ВЕЩЕСТВ;
                                                    - - см. 3.6
function „*" (ЛЕВЫЙ, ПРАВЫЙ: МАТРИЦА) return МАТРИЦА; - - см. 3.6
```

Примеры параметров с выражениями по умолчанию:

```
procedure ПЕЧАТЬ_ЗАГОЛОВКА (СТРАНИЦЫ: in NATURAL;
        ЗАГОЛОВОК: in СТРОЧКА: = (1..СТРОЧКА' LAST =>" );
        ЦЕНТР: in BOOLEAN: = TRUE); - - см. 3.6
```

Примечание. Вычисление выражений по умолчанию начинается при определенных вызовах подпрограмм, как пояснено в разд. 6.4.2 (выражения по умолчанию не вычисляются при предвыполнении описания подпрограммы).

Все подпрограммы могут быть вызваны рекурсивно и являются реентерабельными.

6.2. Виды формальных параметров

Говорят, что значение объекта *читается*, когда это значение вычисляется; оно также читается, когда читается один из его подкомпонентов. Говорят, что значение переменной *изменяется*, когда выполняется присваивание этой переменной, а также (косвенно) когда эта переменная используется в качестве фактического параметра оператора вызова подпрограммы или оператора вызова входа, которые изменяют ее значение; говорят также, что оно изменяется, когда изменяется один из его подкомпонентов.

Виды формальных параметров подпрограммы приведены в табл. 6.1.

Таблица 6.1

Вид	Пояснение
in	Формальный параметр – константа, разрешается только чтение значения соответствующего фактического параметра
in out	Формальный параметр – переменная, разрешается как чтение, так и изменение значения соответствующего фактического параметра
out	Формальный параметр – переменная, разрешается изменение значения соответствующего фактического параметра. Значение скалярного параметра, которое не изменяется при вызове, после возврата не определено; то же самое имеет место для значения скалярного подкомпонента, отличного от дискриминанта. Допускается чтение границ и дискриминантов формального параметра и его подкомпонентов; никакое другое чтение не допускается

Для скалярного параметра такой эффект достигается копированием: в начале каждого вызова значение фактического параметра, соответствующего формальному параметру вида **in** или **in out**, копируется в этом формальном параметре (прямое копирование); затем после нормального окончания тела подпрограммы значение формального параметра вида **in out** или **out** копируется обратно в соответствующем фактическом параметре (обратное копирование). Для параметра ссылочного типа прямое копирование используется для всех трех видов, а обратное – для видов **in out** и **out**.

Для параметров индексируемого, именуемого или задачного типов реализация может достигнуть такого же эффекта копированием, как и для скалярных типов. Кроме того, если копирование используется для параметра вида **out**, то прямое копирование требуется по крайней мере для границ и дискриминантов фактического параметра и его подкомпонентов, а также для каждого подкомпонента ссылочного типа. Другой вариант – вызов ссылкой, когда каждое использование формального параметра (чтение или изменение его значения) рассматривается как использование соответствующего фактического параметра при выполнении вызова подпрограммы. В языке не определяется, какой из этих двух механизмов следует применять для передачи параметров; не определяется также, что различные вызовы одной и той же подпрограммы должны использовать один и тот же механизм. Выполнение программы ошибочно, если ее результат зависит от механизма, выбираемого реализацией.

Для параметра личного типа вышеуказанный эффект достигается по правилу, которое применяется к соответствующему полному описанию типа.

В теле подпрограммы формальный параметр отвечает любому ограничению, вытекающему из обозначения типа, данного в спецификации этого параметра. В качестве границ формального параметра неограниченного индексируемого типа берутся границы фактического параметра (см. 3.6.1). Для формального параметра, описание которого задает неограниченный (личный или именуемый) тип с дискриминантами, дискриминанты этого формального параметра инициализируются значениями соответствующих дискриминантов фактического параметра; формальный параметр неограничен тогда и только тогда, когда его вид **in out** или **out**, и имя переменной, являющейся фактическим параметром, обозначает неограниченную переменную (см. 3.7.1 и 6.4.1).

Если фактический параметр вызова подпрограммы является подкомпонентом, который зависит от дискриминантов переменной неограниченного именуемого типа, то выполнение вызова ошибочно, если значение любого дискриминанта переменной изменяется при этом выполнении; это правило не применяется, если вид параметра **in** и тип подкомпонента – скалярный тип или ссылочный тип.

Примечание. Из правил передачи параметров индексируемого и именуемого типов следует:

• Если выполнение подпрограммы прекращено в результате возбуждения исключения, конечное значение фактического параметра такого типа может быть либо тем

же, что и до вызова, либо значением, присвоенным формальному параметру во время выполнения подпрограммы.

• Если доступ ко всем фактическим параметрам такого типа осуществляется одним способом, то результат вызова подпрограммы (при отсутствии исключения) не зависит от того, использует ли реализация для передачи параметров копирование. Если, однако, доступ к фактическому параметру осуществляется несколькими способами (например, если глобальная переменная или другой формальный параметр ссылается на один и тот же фактический параметр), то значение формального параметра после изменения фактического способом, отличным от изменения формального, неопределенно. Программа, использующая такое неопределенное значение, является ошибочной.

Такие же виды параметров определены и для формальных параметров входов (см. 9.5) с тем же смыслом, что и для подпрограмм. Для формальных параметров настройки определены другие виды параметров (см. 12.1.1).

Для всех видов справедливо, что если фактический параметр указывает задачу, то соответствующий формальный параметр указывает ту же задачу; то же самое имеет место для подкомпонента фактического параметра и соответствующего подкомпонента формального параметра.

6.3. Тела подпрограмм

Тело подпрограммы определяет ее выполнение

```
тело_подпрограммы : : =
 спецификация_подпрограммы is
     [раздел_описаний]
 begin
     последовательность_операторов
 [exception
     обработчик_исключения
     {обработчик_исключения}]
 end [обозначение] ;
```

Описание подпрограммы необязательно. При отсутствии описания спецификация подпрограммы в ее теле (или в следе тела) играет роль описания. Для каждого описания подпрограммы должно быть соответствующее *ему тело (кроме подпрограмм, написанных на другом языке, как поясняется* в разд. 13.9). Если даны и описание, и тело, то спецификация подпрограммы в теле должна быть согласована со спецификацией подпрограммы в описании (см. разд. 6.3.1 о правилах согласования).

Если в конце тела подпрограммы присутствует обозначение, то оно должно совпадать с обозначением в спецификации подпрограммы.

Предвыполнение тела подпрограммы не имеет никакого другого эффекта, кроме установления факта, что тело может быть использовано для выполнения вызовов подпрограммы.

Выполнение тела подпрограммы инициируется вызовом подпрограммы (см. 6.4). Для этого, после установления соответствия между формальными и фактическими параметрами, предвыполняется раздел описаний тела и выполняется последовательность операторов тела подпрограммы. По окончании выполнения тела осуществляется возврат в место вызова (и необходимое обратное копирование значений формальных параметров в фактические (см. 6.2)). Необязательные обработчики исключений, заданные в конце

тела подпрограммы, выполняются при возбуждении исключений во время выполнения последовательности операторов тела подпрограммы (см. 11.4).

Примечание. Из правил видимости следует, что если описанная в пакете подпрограмма обязана быть видимой вне пакета, то спецификация подпрограммы должна быть дана в видимой части пакета. Эти же правила предписывают, что описание подпрограммы должно быть дано, если вызов подпрограммы возникает текстуально до тела подпрограммы (описание должно помещаться в тексте программы раньше вызова). Данные в разд. 3.9 и 7.1 правила подразумевают, что описание подпрограммы и соответствующее ему тело должны находиться непосредственно в одной и той же зоне описаний.

Пример тела подпрограммы:

```
procedure ПРОТАЛКИВАНИЕ (E: in ТИП_ЭЛЕМЕНТА; C: in out СТЕК) is
begin
  if C.ИНДЕКС = C.РАЗМЕР then
     raise ПЕРЕПОЛНЕНИЕ_СЕТКА;
  else
     C.ИНДЕКС:= C.ИНДЕКС + 1;
     C.МЕСТО (C.ИНДЕКС) : = E;
  end if;
end ПРОТАЛКИВАНИЕ;
```

6.3.1. Правила согласования

Всякий раз, когда правила языка требуют или допускают появления спецификации данной подпрограммы более одного раза, в каждом месте допустимы следующие вариации:

• Числовой литерал может быть заменен другим числовым литералом тогда и только тогда, когда они имеют одно и то же значение.

• Простое имя может быть заменено расширенным именем, в котором это простое имя является постфиксом тогда и только тогда, когда смысл простого имени в обоих случаях определяется одним и тем же описанием.

• Строковый литерал в качестве знака операции может быть заменен на другой строковый литерал тогда и только тогда, когда они представляют одну и ту же операцию (см. 8.5).

Две спецификации подпрограммы называются *согласованными*, если, за исключением комментариев и приведенных выше вариаций, обе спецификации образованы одной и той же последовательностью лексем, и соответствующие лексемы имеют одинаковый смысл с точки зрения правил видимости и совмещения.

Аналогично определяется согласование для разделов формальных параметров, разделов дискриминантов и обозначений типов (для субконстант и фактических параметров, которые имеют форму преобразования типа (см. 6.4.1)).

Примечание. Простое имя может быть заменено на расширенное имя, даже если простое имя само является префиксом именуемого компонента. Например, С.Р может быть заменено на К.С.Р, если С описано непосредственно в К.

Следующие спецификации не согласуются, так как они сформированы различными последовательностями лексем:

```
procedure P (X, Y: INTEGER)
```

procedure P (X: INTEGER; Y: INTEGER)
procedure P (X, Y: **in** INTEGER)

6.3.2. Подстановка подпрограммы

Прагма INLINE используется для указания того, что желательна прямая замена телом подпрограммы каждого вызова каждой поименованной в прагме подпрограммы. Форма этой прагмы следующая:

pragma INLINE (имя {, имя});

Каждое имя – это либо имя подпрограммы, либо имя настраиваемой подпрограммы. Прагма INLINE допустима только на месте элемента описания в разделе описаний или спецификации пакета, либо после библиотечного модуля в компиляции, но до любого следующего компилируемого модуля.

Если прагма стоит на месте элемента описания, то каждое имя должно обозначать подпрограмму или настраиваемую подпрограмму, описанные раньше в виде элемента описания этого же раздела описаний или этой же спецификации пакета. Если несколько (совмещенных) подпрограмм удовлетворяют этому требованию, то прагма применяется ко всем подпрограммам. Если эта прагма стоит после данного библиотечного модуля, то в качестве ее аргумента допустимо только имя этого модуля. Если в прагме упомянуто имя настраиваемой подпрограммы, это указывает, что подстановка желательна для вызовов всех подпрограмм, являющихся конкретизацией поименованного настраиваемого модуля.

Смысл подпрограммы не изменяется прагмой INLINE. Для каждого вызова поименованных подпрограмм реализация может выполнять или игнорировать рекомендации прагмы. (Заметим, в частности, что подстановка не может быть выполнена для рекурсивных подпрограмм).

6.4. **Вызовы подпрограмм**

Вызов подпрограммы – это либо оператор вызова процедуры, либо вызов функции; он вызывает выполнение соответствующего тела подпрограммы. Вызов определяет связь фактических параметров, если они есть, с формальными параметрами подпрограммы.

оператор_вызова_процедуры : : = имя_*процедуры*
  [раздел_фактических_параметров] ;
вызов_функции : : = имя_*функции* [раздел_фактических_параметров]
раздел_фактических_параметров : : =
  (сопоставление_параметров {, сопоставление_параметров})
сопоставление_параметров : : =
  [формальный_параметр =>] фактический_параметр
формальный_параметр : : = простое_имя_*параметра*
фактический_параметр : : = выражение | имя_*переменной*
  | обозначение_типа (имя_*переменной*)

Каждое сопоставление параметров связывает фактический параметр с соответствующим формальным параметром. Сопоставление параметров называется *именованным,* если формальный параметр указан явно, в противном случае оно называется *позиционным.* Для позиционного сопоставления фак-

тический параметр соответствует формальному параметру в той же позиции раздела формальных параметров.

Именованные сопоставления могут быть даны в любом порядке, но если в одном и том же вызове использованы позиционные и именованные сопоставления, то позиционные сопоставления должны стоять первыми, на своих местах. Следовательно, после именованного сопоставления все остальные должны быть только именованными сопоставлениями.

Для каждого формального параметра подпрограммы вызов подпрограммы должен задавать точно один соответствующий фактический параметр. Этот фактический параметр определяется либо явно сопоставлением параметра, либо, в отсутствие такого сопоставления, выражением по умолчанию (см. 6.4.2).

Сопоставления параметров вызова подпрограммы вычисляются в некотором порядке, не определенном в языке. Аналогично, правила языка не определяют, в каком порядке значения параметров вида **in out** или **out** копируются обратно в соответствующих фактических параметрах (если это делается).

Примеры вызовов процедур:

```
ОБХОД_ДЕРЕВА;   - см. 6.1
УПРАВЛЕНИЕ_ТАБЛИЦЕЙ.ВСТАВКА (Е) ; - - см. 7.5
ПЕЧАТЬ_ЗАГОЛОВКА (128, ТИТУЛ, TRUE); - - см. 6.1
ПЕРЕКЛЮЧАТЕЛЬ (ИЗ => X, В => СЛЕДУЮЩИЙ; - - см. 6.1
ПЕЧАТЬ_ЗАГОЛОВКА (128, ЗАГОЛОВОК => ТИТУЛ, ЦЕНТР => TRUE);
        - - см. 6.1
ПЕЧАТЬ_ЗАГОЛОВКА (ЗАГОЛОВОК => ТИТУЛ, ЦЕНТР => TRUE,
        СТРАНИЦЫ => 128) ; - - см. 6.1
```

Примеры вызовов функций:

```
СКАЛ_ПРОИЗВЕДЕНИЕ (А, В) - - см. 6.1 и 6.5
CLOCK - - см. 9.6
```

6.4.1. Сопоставления параметров

Тип каждого фактического параметра должен совпадать с типом соответствующего формального параметра.

Фактический параметр, сопоставляемый с формальным параметром вида **in**, должен быть выражением; оно вычисляется до вызова.

Фактический параметр, сопоставляемый с формальным параметром вида **in out** или **out**, должен быть либо именем переменной, либо иметь форму преобразования типа с аргументом, являющимся именем переменной. В любом случае, для параметра вида **in out** переменная не должна быть формальным параметром вида **out** или подкомпонентом такого параметра. Для фактического параметра, который имеет форму преобразования типа, обозначение типа должно быть согласовано (см. 6.3.1) с обозначением типа формального параметра; допустимый операнд и целевой тип такие же, как и для преобразования типа (см. 4.6).

Данное для фактического параметра вида **in out** или **out** имя переменной вычисляется до вызова. Если фактический параметр имеет форму преобразования типа, то перед вызовом для параметра вида **in out** переменная преобразуется к заданному типу; после (нормального) окончания тела под-

программы формальные параметры вида **in out** или **out** преобразуются обратно в тип переменной. (Тип преобразования должен быть тем же, что и у формального параметра).

Для параметров скалярного и ссылочного типов проверяются следующие ограничения:

• Перед вызовом для параметра вида **in** или **in out** проверяется принадлежность фактического параметра подтипу формального параметра.

• После (нормального) окончания тела подпрограммы: для параметра вида **in out** или **out** проверяется принадлежность значения формального параметра подтипу фактического параметра. В случае преобразования типа значение формального параметра преобразуется обратно, и проверяется результат преобразования.

В каждом из вышеуказанных случаев выполнение программы ошибочно, если проверяемое значение неопределенно.

Для параметров других типов проверка делается до вызова для всех видов, как для скалярных и ссылочных типов; после возврата никаких проверок не делается.

Если результат хотя бы одной проверки отрицателен, при вызове подпрограммы возбуждается исключение CONSTRAINT_ERROR.

Примечание. Если обозначение типа формального параметра задает ограниченный подтип, то для индексируемых типов и типов с дискриминантами достаточно проверки перед вызовом (проверка после возврата была бы избыточной), так как ни границы массива, ни дискриминанты не могут быть изменены.

Если это обозначение типа задает неограниченный индексируемый тип, то формальный параметр ограничен границами соответствующего фактического параметра, и никакой проверки не требуется (ни до вызова, ни после возврата, см. 3.6.1). Аналогично, не требуется никакой проверки, если обозначение типа обозначает неограниченный тип с дискриминантами, так как формальный параметр ограничен точно так же, как соответствующий фактический параметр (см. 3.7.1).

6.4.2. Опущенные параметры

Если спецификация параметра включает выражение по умолчанию для параметра вида **in**, то соответствующие вызовы подпрограммы не обязательно содержат сопоставления для такого параметра. Если в вызове сопоставления для таких параметров опускается, то оставшаяся часть вызова, следующая за начальными позиционными сопоставлениями, должна использовать только именованные сопоставления.

Для любого опущенного сопоставления параметров выражение по умолчанию вычисляется до вызова, а значение результата используется как неявный фактический параметр.

Примеры процедур со значениями по умолчанию:

```
procedure АКТИВИЗИРОВАТЬ (ПРОЦЕСС: in ИМЯ_ПРОЦЕССА;
        ПОСЛЕ: in ИМЯ_ПРОЦЕССА: = НЕТ_ПРОЦЕССА;
        ЖДАТЬ: in DURATION: = 0.0;
        ПРИОР: in BOOLEAN: = FALSE);
procedure ПАРА (ЛЕВЫЙ, ПРАВЫЙ: ИМЯ_ПЕРСОНЫ: = new ПЕРСОНА);
```

Примеры их вызовов:

```
АКТИВИЗИРОВАТЬ (Х);
```

```
АКТИВИЗИРОВАТЬ (X, ПОСЛЕ => Y);
АКТИВИЗИРОВАТЬ (X, ЖДАТЬ => 60.0, ПРИОР => TRUE);
АКТИВИЗИРОВАТЬ (X, Y, 10.0, FALSE);
ПАРА;
ПАРА (ЛЕВЫЙ => new ПЕРСОНА, ПРАВЫЙ => new ПЕРСОНА);
```

Примечание. Если выражение по умолчанию используется для двух или более параметров в групповой спецификации параметров, то это выражение по умолчанию вычисляется один раз для каждого опущенного параметра. Поэтому в примере два вызова процедуры ПАРА эквивалентны.

6.5. **Функции**

Функция – это подпрограмма, которая возвращает значение (результат вызова функции). Спецификация функции начинается с зарезервированного слова **function**, а параметры, если они есть, должны иметь вид **in** (указанный явно или неявно). Операторы тела функции (исключая операторы программных модулей, вложенных в тело функции) должны содержать один или несколько операторов возврата, определяющих возвращаемое значение.

Исключение PROGRAM_ERROR возбуждается, если выход из тела функции осуществляется не через оператор возврата. Это исключение не возбуждается, если выполнение функции прекращается в результате исключения.

Пример:

```
function СКАЛ_ПРОИЗВЕДЕНИЕ (ЛЕВЫЙ, ПРАВЫЙ: ВЕКТОР)
        return ВЕЩЕСТВ is СУММА: ВЕЩЕСТВ: = 0.0;
begin ПРОВЕРКА (ЛЕВЫЙ' FIRST = ПРАВЫЙ' FIRST and ЛЕВЫЙ' LAST =
                = ПРАВЫЙ'LAST);
  for K in ЛЕВЫЙ'RANGE loop
     СУММА: = СУММА + ЛЕВЫЙ (К) * ПРАВЫЙ (К);
  end loop;
  return СУММА;
end СКАЛ_ПРОИЗВЕДЕНИЕ;
```

6.6. **Профиль типа параметров и результата. Совмещение подпрограмм**

Два раздела формальных параметров называются имеющими одинаковый *профиль типа параметров* тогда и только тогда, когда они имеют одинаковое число параметров, а в каждой позиции соответствующие параметры имеют один и тот же базовый тип. Подпрограмма или вход имеет одинаковый *профиль типа параметров и результата* с другой подпрограммой или входом тогда и только тогда, когда оба имеют одинаковый профиль типа параметров, и либо оба являются функциями с одним и тем же базовым типом результата, либо оба функциями не являются.

Один и тот же идентификатор подпрограммы или знак операции может быть использован для нескольких спецификаций подпрограмм. В этом случае идентификатор или знак операции называется *совмещенным;* подпрограммы, которые имеют этот идентификатор или знак операции, тоже называются совмещенными и совмещаются друг с другом. Как поясняется в разд. 8.3, если две подпрограммы совмещаются друг с другом, то одна из них может скрыть другую, только если обе подпрограммы имеют одинаковый профиль типа параметров и результата и разные зоны описания (см. 8.3,

где описаны другие требования, которые должны быть удовлетворены для скрытия).

Вызов совмещенной подпрограммы неоднозначен (и поэтому неправилен), если ее имя, число сопоставлений параметров, типы и порядок фактических параметров, имена формальных параметров (при использовании именованных сопоставлений параметров) и тип результата (для функций) не позволяют идентифицировать единственную (совмещенную) спецификацию подпрограммы.

Примеры совмещенных подпрограмм:

```
procedure PUT (X: INTEGER);
procedure PUT (X: STRING);
procedure SET (КОЛОРИТ: ЦВЕТ);
procedure SET (СИГНАЛ: СВЕТ);
```

Примеры вызовов:

```
PUT (28);
PUT ("здесь нет возможной неоднозначности");
SET (КОЛОРИТ => КРАСНЫЙ);
SET (СИГНАЛ => КРАСНЫЙ);
SET (ЦВЕТ'КРАСНЫЙ);
-- SET (КРАСНЫЙ) может быть неоднозначным, так как
-- КРАСНЫЙ может обозначать значение типа ЦВЕТ и типа СВЕТ.
```

Примечание. Понятие профиля типа параметров и результата не учитывает имен параметров, их видов и подтипов, а также присутствия или отсутствия выражений по умолчанию.

Неоднозначности могут (но не обязательно) возникнуть, когда фактические параметры вызова совмещенной подпрограммы сами являются вызовами совмещенной функции, совмещенными литералами или агрегатами. Неоднозначности могут (но не обязательно) также возникнуть, когда видимы несколько совмещенных подпрограмм, принадлежащих различным пакетам. Этих неоднозначностей можно избежать несколькими способами: можно использовать квалифицированные выражения для некоторых или всех фактических параметров и результата, если он есть; имя такой подпрограммы можно задавать более точно расширенным именем; наконец, такая подпрограмма может быть переименована.

6.7. Совмещение операций

Описание функции, обозначение которой является знаком операции, используется для совмещения операций. Последовательность символов в знаке операции должна быть обозначением операций: логической, отношения, бинарной аддитивной, унарной аддитивной, мультипликативной или высшего приоритета (см. 4.5). В качестве обозначения функции не допускаются никакие проверки вхождения, ни формы управления с промежуточной проверкой.

Спецификация подпрограммы унарной операции должна иметь один единственный параметр. Спецификация подпрограммы бинарной операции должна иметь два параметра; при каждом использовании этой операции левый операнд берется в качестве первого фактического параметра, правый операнд – в качестве второго параметра; конкретизация настройки функции, которая обозначена знаком операции, допускается, только если спецификация настраиваемой функции имеет соответствующее число парамет-

ров. Выражения по умолчанию для параметров операции недопустимы (описана ли операция явно спецификацией подпрограммы или конкретизацией настройки).

Операции "+" и "–" допускают как унарную, так и бинарную совмещенную операцию.

Явное описание функции, которая совмещает операцию "=", отличное от описания переименования, допустимо только если оба параметра являются параметрами одного и того же лимитируемого типа. Совмещение равенства должно давать результат предопределенного типа BOOLEAN; операция неравенства "/=", дающая результат, дополнительный к результату операции равенства, совмещается неявно при задании операции равенства. Явное совмещение операции неравенства недопустимо.

Описание переименования, обозначение которого – операция равенства, допустимо только для переименования другой операции равенства. (Например, такое описание переименования может быть использовано, когда равенство видимо по имени, но не непосредственно).

Примечание. Совмещение операций отношения не нарушает соотношений, таких как проверка принадлежности диапазону или выборы в операторе выбора.

Примеры:

```
function "+" (ЛЕВЫЙ, ПРАВЫЙ: МАТРИЦА) return МАТРИЦА;
function "+" (ЛЕВЫЙ, ПРАВЫЙ: ВЕКТОР) return ВЕКТОР;
  - - в предположении, что A, B и C типа ВЕКТОР,
  - - три следующих присваивания эквивалентны
A: = B + C;
A: = "+" (B, C);
A: = "+" (ЛЕВЫЙ => B, ПРАВЫЙ=> C);
```

7. ПАКЕТЫ

Пакеты – это одна из четырех форм программных модулей, из которых составляются программы. Другие формы – это подпрограммы, задачные модули и настраиваемые модули.

Пакеты допускают спецификацию групп логически связанных понятий. Простейшие формы пакета задают совокупности общих объектов и описаний типов. В более общем случае пакеты могут использоваться для задания групп взаимосвязанных понятий, включающих также подпрограммы, которые могут быть вызваны вне пакета, тогда как внутренняя работа остается скрытой и защищенной от внешних пользователей.

7.1. Структура пакета

Пакет обычно представлен двумя частями: спецификацией пакета и телом пакета. Спецификация имеется у каждого пакета, а тело пакета имеют не все пакеты.

```
описание_пакета : : = спецификация_пакета;
спецификация_пакета : : =
  package идентификатор is
```

```
    {основной_элемент_описания}
  [private
    {основной_элемент_описания}]
 end [простое_имя_пакета]
тело_пакета : : =
 package body простое_имя_пакета is
    [раздел_описаний]
 [begin
    последовательность_операторов
 [exception
    обработчик_исключения
    {обработчик_исключения}] ]
 end [простое_имя_пакета] ;
```

Простое имя в начале тела пакета должно повторять идентификатор этого пакета. Аналогично, если простое имя помещено в конце спецификации или тела пакета, то оно должно повторять идентификатор этого пакета.

Если описание подпрограммы, описание пакета, описание задачи или описание настройки является элементом описания в спецификации пакета, то тело (если оно существует) программного модуля, описанного этим элементом описания, само должно быть элементом описания в разделе описаний тела того же самого пакета.

Примечание. Для простой формы пакета, задающей совокупность объектов и типов, тело не обязательно. Одной из возможностей использования последовательности операторов тела пакета является инициализация таких объектов. Для каждого описания подпрограммы должно существовать соответствующее ему тело (за исключением подпрограмм, написанных на другом языке, см. разд. 13.9). Если тело программного модуля является следом тела, то для этого программного модуля требуется раздельно компилируемый субмодуль, содержащий соответствующее тело (см. 10.2). Тело не является основным элементом описания и, таким образом, не может присутствовать в спецификации пакета.

Описание пакета – это либо библиотечный пакет (см. 10.2), либо элемент описания внутри другого программного модуля.

7.2. Спецификации и описание пакетов

Первый список элементов описания в спецификации пакета называется *видимым разделом* пакета. Необязательный список элементов описания после зарезервированного слова **private** называется *личным разделом* пакета.

Понятие, описанное в личном разделе пакета, не видимо вне этого пакета (имя, обозначающее такое понятие, доступно только в пакете). В противоположность этому расширенные имена, обозначающие описанные в видимом разделе понятия, могут быть использованы даже вне этого пакета; прямую видимость этих понятий можно получить также с помощью спецификатора использования (см. 4.1.3 и 8.4).

Предвыполнение описания пакета состоит в предвыполнении его основных элементов описания в порядке их следования.

Примечание. Видимый раздел пакета содержит всю информацию, доступную для другого программного модуля. Пакет, состоящий только из спецификации пакета

(т. е. без тела пакета), может быть использован для представления группы общих констант или переменных, или общей совокупности объектов и типов, как показано ниже в примерах.

Пример пакета, описывающего группу общих переменных:

```
package ГРАФИЧЕСКИЕ_ДАННЫЕ is
  ПЕРО_В: BOOLEAN;
  КОЭФФИЦИЕНТ_ПЕРЕСЧЕТА,
  X_СМЕЩЕНИЕ, Y_СМЕЩЕНИЕ,
  X_МИН, Y_МИН, X_МАКС, Y_МАКС: ВЕЩЕСТВ; -- см. 3.5.7
  X_ЗНАЧЕНИЕ: array (1..500) of ВЕЩЕСТВ;
  Y_ЗНАЧЕНИЕ: array (1..500) of ВЕЩЕСТВ;
end ГРАФИЧЕСКИЕ_ДАННЫЕ;
```

Пример пакета, описывающего общую совокупность объектов и типов:

```
package РАБОЧИЕ_ДНИ is
  type ДЕНЬ is (ПНД, ВТР, СРД, ЧТВ, ПТН, СБТ, ВСК);
  type РАБОЧИЕ_ЧАСЫ is delta 0.25 range 0.0..24.0;
  type ТАБЕЛЬ_ВРЕМЕНИ is array (ДЕНЬ) of
     РАБОЧИЕ_ЧАСЫ;
  ЧАСЫ_РАБОТЫ: ТАБЕЛЬ_ВРЕМЕНИ;
  ОБЫЧНЫЕ_ЧАСЫ: constant ТАБЕЛЬ_ВРЕМЕНИ : =
     (ПНД..ЧТВ => 8.25, ПТН => 7.0, СБТ | ВСК => 0.0);
end РАБОЧИЕ_ДНИ;
```

7.3. Тела пакетов

В отличие от понятий, описанных в видимом разделе спецификации пакета, понятия, описанные в теле пакета, видимы только внутри самого тела пакета. Поэтому пакет с телом пакета может быть использован для создания группы взаимосвязанных подпрограмм (пакет прикладных программ в обычном смысле), в которой доступные пользователям операции явно изолированы от внутренних понятий.

При предвыполнении тела пакета сначала предвыполняется его раздел описаний, а затем выполняется его последовательность операторов (если она имеется). Необязательно присутствующие в конце тела пакета обработчики исключений обслуживают исключения, возбуждаемые при выполнении последовательности операторов тела пакета.

Примечание. Переменная, описанная в теле пакета, видима только внутри этого тела, и, следовательно, ее значение может быть изменено только внутри этого тела пакета. В отсутствии локальных задач, значение такой переменной сохраняется неизменным между вызовами извне пакета подпрограмм, описанных в его видимом разделе. Свойства такой переменной аналогичны свойствам „собственной" переменной в языке Алгол 60.

Предвыполнение тела подпрограммы, описанной в видимом разделе пакета, осуществляется при предвыполнении тела пакета. Следовательно, при вызове такой подпрограммы извне программного модуля возбуждается исключение PROGRAM_ERROR, если вызов производится до предвыполнения тела пакета (см. 3.9).

Пример пакета:

```
package РАЦИОНАЛЬНЫЕ_ЧИСЛА is
  type РАЦИОНАЛЬНЫЙ is
     record
        ЧИСЛИТЕЛЬ: INTEGER,
        ЗНАМЕНАТЕЛЬ: POSITIVE;
```

```
    end record;
  function РАВНО (X, Y: РАЦИОНАЛЬНЫЙ) return BOOLEAN;
  function "/" (X, Y: INTEGER) return РАЦИОНАЛЬНЫЙ;
    - - для образования рационального числа
  function "+" (X, Y: РАЦИОНАЛЬНЫЙ) return РАЦИОНАЛЬНЫЙ;
  function "–" (X, Y: РАЦИОНАЛЬНЫЙ) return РАЦИОНАЛЬНЫЙ;
  function "*" (X, Y: РАЦИОНАЛЬНЫЙ) return РАЦИОНАЛЬНЫЙ;
  function "/" (X, Y: РАЦИОНАЛЬНЫЙ) return РАЦИОНАЛЬНЫЙ;
end;
package body РАЦИОНАЛЬНЫЕ_ЧИСЛА is
  procedure ОБЩИЙ_ЗНАМЕНАТЕЛЬ (X, Y: in out РАЦИОНАЛЬНЫЙ) is
  begin
  - - приведение X и Y к общему знаменателю:
    . . .
  end;
  function РАВНО (X, Y: РАЦИОНАЛЬНЫЙ) return BOOLEAN is
    A, B: РАЦИОНАЛЬНЫЙ;
  begin
    A: = X;
    B: = Y;
    ОБЩИЙ_ЗНАМЕНАТЕЛЬ (A, B);
    return A. ЧИСЛИТЕЛЬ = B.ЧИСЛИТЕЛЬ;
end РАВНО;
function "/" (X, Y: INTEGER) return РАЦИОНАЛЬНЫЙ is
begin
  if Y > 0 then
    return (ЧИСЛИТЕЛЬ => X, ЗНАМЕНАТЕЛЬ => Y);
  else
    return (ЧИСЛИТЕЛЬ => –X, ЗНАМЕНАТЕЛЬ => –Y);
  end if;
end "/";
function "+" (X, Y: РАЦИОНАЛЬНЫЙ) return РАЦИОНАЛЬНЫЙ is . . .end "+";
function "–" (X, Y: РАЦИОНАЛЬНЫЙ) return РАЦИОНАЛЬНЫЙ is. . .end "–";
function "*" (X, Y: РАЦИОНАЛЬНЫЙ) return РАЦИОНАЛЬНЫЙ is. . .end "*";
function "/" (X, Y: РАЦИОНАЛЬНЫЙ) return РАЦИОНАЛЬНЫЙ is. . .end "/";
end РАЦИОНАЛЬНЫЕ_ЧИСЛА;
```

7.4. Описания личных типов и субконстант

Описание типа в качестве личного (приватного) в видимом разделе пакета служит для отделения характеристик, которые могут быть использованы непосредственно внешними программными модулями (логические свойства), от других характеристик, непосредственное использование которых возможно только внутри пакета (детали определения самого типа). Описания субконстант задают константы личных типов.

описание_личного_типа : : =

type идентификатор [раздел_дискриминантов] **is [limited] private;**

описание_субконстанты : : =

список_идентификаторов : **constant** обозначение_типа;

Описание личного типа допустимо только в качестве элемента описания в видимом разделе пакета или в качестве описания параметра настройки для типа в разделе формальных параметров настройки.

Обозначение типа в описании субконстанты должно обозначать личный тип или подтип личного типа. Описание субконстанты и описание соответст-

вующего личного типа должны быть оба элементами описаний в видимом разделе одного и того же пакета. Описание субконстант с несколькими идентификаторами эквивалентно последовательности единичных описаний субконстант (см. 3.2).

Примеры описаний личного типа:

```
type КЛЮЧ is private;
type ИМЯ_ФАЙЛА is limited private;
```

Пример описания субконстанты:

```
НУЛЕВОЙ_КЛЮЧ. constant КЛЮЧ;
```

7.4.1. Личные типы

Если описание личного типа дается в видимом разделе пакета, то соответствующее описание типа с тем же самым идентификатором должно присутствовать в качестве элемента описания в личном разделе пакета. Соответствующее описание должно быть либо полным описанием типа, либо описанием задачного типа. В оставшейся части этого раздела объяснения даются для полных описаний типов. Те же правила применяются к описаниям задачных типов.

Описание личного типа и соответствующее полное описание типа определяют один тип. Описание личного типа вместе с видимым разделом определяют операции, которые могут использовать внешние программные модули (см. разд. 7.4.2 об операциях, применимых к личному типу). С другой стороны, полное описание типа определяет другие операции, непосредственное использование которых возможно только внутри самого пакета.

Если описание личного типа включает раздел дискриминантов, то полное описание типа должно включать раздел дискриминантов по правилам согласования (см. 6.3.1), и определением типа должно быть определение именуемого типа. И наоборот, если описание личного типа не включает раздел дискриминантов, то тип, описанный с помощью полного описания типа (*полный* тип), не должен быть неограниченным типом с дискриминантами. Полный тип не должен быть неограниченным индексируемым типом. Лимитируемый тип (в частности, задачный тип) допускается в качестве полного типа, только если в описании личного типа присутствует зарезервированное слово **limited** (см. 7.4.4).

Внутри спецификации пакета, в которой описан личный тип, и до конца соответствующего полного описания типа ограничено использование имени этого личного типа или его подтипа, а также имени любого типа или подтипа с подкомпонентом данного личного типа. Использование такого имени допускается только в описании субконстанты, описании типа или подтипа, спецификации подпрограммы или описания входа; более того, не допускается использование этого имени в определениях производного типа или в простых выражениях.

При предвыполнении описания личного типа создается личный тип. Если описание личного типа имеет раздел дискриминантов, то он также предвыполняется. Предвыполнение полного описания типа заключается в предвыполнении определения типа; если имеется раздел дискриминантов, то он

не предвыполняется (так как уже предвыполнен согласованный раздел дискриминантов в описании личного типа).

Примечание. Из перечисленных правил следует, что ни описание переменной личного типа, ни создание объекта личного типа с помощью генератора недопустимо до полного описания типа. Точно так же до полного описания нельзя использовать имя личного типа в конкретизации настройки или в спецификаторе представления.

7.4.2. Операции над личным типом

Операции над личным типом, которые неявно описаны, включают базовые операции: присваивание (кроме лимитируемого типа), проверку принадлежности, квалификацию, явное преобразование и именование компоненты для именования любого дискриминанта.

Для личного типа T базовые операции также включают атрибуты T'BASE (см. 3.3.3) и T'SIZE (см. 13.7.2). Для объекта A личного типа базовые операции включают атрибут A'CONSTRAINED, если личный тип содержит дискриминант (см. 3.7.4), и во всех случаях – атрибуты A'SIZE и A'ADDRESS (см. 13.7.2).

Наконец, в операции, неявно описанные описанием личного типа, включаются предопределенные отношения равенства и неравенства (кроме личных типов, в описании которых присутствует зарезервированное слово **limited**).

Рассмотренные выше операции вместе с подпрограммами, которые имеют параметр или результат личного типа и которые описаны в видимом разделе пакета, являются единственными операциями над личным типом, используемыми вне пакета.

Внутри пакета, содержащего описание личного типа, дополнительные операции вводятся неявно полным описанием типа. При этом переопределение этих операций допустимо внутри той же самой зоны описаний, в том числе и между описанием личного типа и соответствующим полным описанием. Явно описанная подпрограмма скрывает неявно описанную операцию, которая имеет тот же самый профиль типа параметров и результата (это возможно только, если неявно описанной операцией является производная подпрограмма или предопределенная операция).

Если составной тип имеет подкомпонент личного типа и описан вне пакета, содержащего описание этого личного типа, то неявно описанные при описании составного типа операции включают все операции, которые зависят только от характеристик, следующих из одного описания личного типа. (Например, операция $<$ не включается в набор операций для одномерного индексируемого типа).

Если составной тип сам описан внутри пакета, содержащего описание личного типа (включая внутренний пакет или настраиваемый пакет), то неявно описываются дополнительные операции, которые зависят от характеристик полного типа, как этого требуют правила, применимые к составному типу (например, операция $<$ описана для одномерного индексируемого типа, если полный тип является дискретным). Эти дополнительные операции считаются неявно описанными в самом начале непосредственной

области действия составного типа, их использование разрешено только после полного описания типа.

Те же правила относятся к операциям, которые неявно описаны для ссылочного типа, чье указание типа есть личный тип или тип, описанный посредством неполного описания типа.

Для каждого личного типа или подтипа T определен следующий атрибут:

T'CONSTRAINED Вырабатывает значение FALSE, если T обозначает неограниченный личный тип с дискриминантами, не являющийся формальным параметром настройки; вырабатывает также значение FALSE, если T обозначает личный тип, являющийся формальным параметром настройки, а соответствующий подтип фактического параметра является либо неограниченным типом с дискриминантом, либо неограниченным индексируемым типом; в остальных случаях вырабатывает значение TRUE. Значение атрибута имеет предопределенный тип BOOLEAN.

Примечание. Описание личного типа и соответствующее полное описание типа определяют два разных аспекта одного и того же типа. Вне пакета тип обладает теми характеристиками, которые определены в видимой части. Для внешних программных модулей тип является как раз личным типом, и любое правило языка, которое применяется только к другому классу типов, к этому типу не применимо. Тот факт, что полное описание может *реализовать* личный тип в виде типа конкретного класса (например, в виде индексируемого типа), можно использовать только внутри пакета.

Последствия такой фактической реализации сказываются, тем не менее, везде. Например, производится некоторая инициализация компонентов по умолчанию; атрибут SIZE вырабатывает размер полного типа; правила зависимости задач распространяются также на компоненты – объекты задачного типа.

Пример:

```
package УПРАВЛЕНИЕ_ПО_КЛЮЧУ is
  type КЛЮЧ is private;
  НУЛЕВОЙ_КЛЮЧ: constant КЛЮЧ;
  procedure УСТАНОВИТЬ_КЛЮЧ (К: out КЛЮЧ);
  function "<" (X, Y: КЛЮЧ) return BOOLEAN;
private
  type КЛЮЧ is new NATURAL;
  НУЛЕВОЙ_КЛЮЧ: constant КЛЮЧ: = 0;
end;
package body УПРАВЛЕНИЕ_ПО_КЛЮЧУ is
  ПОСЛЕДНИЙ_КЛЮЧ: КЛЮЧ: = 0;
  procedure УСТАНОВИТЬ_КЛЮЧ (К: out КЛЮЧ) is
  begin
     ПОСЛЕДНИЙ_КЛЮЧ: = ПОСЛЕДНИЙ_КЛЮЧ + 1;
     К: = ПОСЛЕДНИЙ_КЛЮЧ;
  end УСТАНОВИТЬ_КЛЮЧ;
  function "<" (X, Y: КЛЮЧ) return BOOLEAN is
  begin
     return INTEGER (X) < INTEGER (Y);
  end "<";
end УПРАВЛЕНИЕ_ПО_КЛЮЧУ;
```

Примечание к примеру. Операциями, применимыми к объектам типа КЛЮЧ вне пакета УПРАВЛЕНИЕ_ПО_КЛЮЧУ, являются: присваивание, сравнение на равенство

и неравенство, процедура УСТАНОВИТЬ_КЛЮЧ и операция "<"; сюда не включаются другие операции отношения, например, ">=" или арифметические операции.

Явно описанная операция "<" скрывает предопределенную операцию "<", неявно описанную полным описанием типа. В теле функции необходимо явное преобразование X и Y к типу INTEGER для явного вызова операции "<" для этого типа. С другой стороны, результат функции мог бы быть записан в виде **not** (X >= Y), так как операция ">=" не переопределена.

Значение переменной ПОСЛЕДНИЙ_КЛЮЧ, описанной в теле пакета, не меняется между вызовами процедуры УСТАНОВИТЬ_КЛЮЧ (см. также примечание в разд. 7.3).

7.4.3. Субконстанты

Если описание субконстанты дается в видимом разделе пакета, то описание константы (т. е. описание объекта, задающее константу с явной инициализацией) с тем же самым идентификатором должно быть элементом описания личного раздела этого пакета. Такое описание объекта называется *полным* описанием субконстанты. Заданное в полном описании обозначение типа должно быть согласовано с обозначением типа, заданным в описании субконстанты (см. 6.3.1). Допускаются групповые и единичные полные описания и описания субконстант при условии, что эквивалентные единичные описания согласованы.

В спецификации пакета, содержащей описание субконстанты, и до конца соответствующего полного описания имя субконстанты допускается использовать только в выражении по умолчанию для именуемого компонента или формального параметра (но не формального параметра настройки).

Предвыполнение описания субконстанты не дает другого эффекта.

Выполнение программы ошибочно, если оно пытается использовать значение субконстанты до предвыполнения соответствующего полного описания.

Примечание. Полное описание субконстанты заданного личного типа не должно встречаться до соответствующего полного описания типа. Это является следствием правил, определяющих допустимые использования имени, обозначающего личный тип (см 7.4.1).

7.4.4. Лимитируемые типы

Лимитируемый тип – это тип, для которого *неявным* описанием не вводится ни присваивание, ни сравнение на равенство и неравенство.

Описание личного типа с зарезервированным словом **limited** описывает лимитируемый тип. Задачный тип является лимитируемым типом. Производный тип от лимитируемого типа сам является лимитируемым типом. Наконец, составной тип является лимитируемым, если тип одного из его компонентов является лимитируемым.

Над личным лимитируемым типом определены операции, которые даны в разд. 7.4.2, за исключением присваивания и предопределенного сравнения на равенство и неравенство.

Вид **out** допустим для формального параметра лимитируемого типа в явно описанной подпрограмме только в том случае, если этот тип является личным лимитируемым типом, а описание подпрограммы находится в видимом разделе пакета, где описан и личный тип. То же самое относится к фор-

мальным параметрам описаний входов и описаний настраиваемых процедур. Соответствующий полный тип не должен быть лимитируемым, если хотя бы один такой формальный параметр имеет вид **out**. В противном случае, в качестве соответствующего полного типа допускается (но не требуется) лимитируемый тип (в частности, допустим задачный тип). Если полный тип, соответствующий лимитируемому типу, сам не является лимитируемым, то для этого типа в пакете (но не вне его) допустимо присваивание.

Из правил для лимитируемых типов вытекает следующее:

• Если тип объекта является лимитируемым, то в описании этого объекта явная инициализация недопустима.

• Если тип именуемого компонента является лимитируемым, то выражение по умолчанию в описании компонента недопустимо.

• Если тип объекта, указанного ссылочным типом, является лимитируемым, то в генераторе явная инициализация недопустима.

• Формальный параметр настройки вида **in** не должен быть лимитируемого типа.

Примечание. Описанные выше правила не исключают выражение по умолчанию для формального параметра лимитируемого типа; они не исключают также субконстанту лимитируемого типа, если полный тип не является лимитируемым. Для лимитируемого типа допускается явное описание операции равенства (см. 6.7).

Для лимитируемого составного типа не разрешаются агрегаты (см. 3.6.2 и 3.7.4). Для лимитируемого индексируемого типа не разрешается катенация (см. 3.6.2)

Пример:

```
package ПАКЕТ_ВВОДА_ВЫВОДА is
  type ИМЯ_ФАЙЛА is limited private;
  procedure ОТКРЫТЬ (F: in out ИМЯ_ФАЙЛА);
  procedure ЗАКРЫТЬ (F: in out ИМЯ_ФАЙЛА);
  procedure ЧИТАТЬ (F: in ИМЯ_ФАЙЛА; ЭЛЕМЕНТ: out INTEGER);
  procedure ПИСАТЬ (F: in ИМЯ_ФАЙЛА; ЭЛЕМЕНТ: in INTEGER);
private
  type ИМЯ_ФАЙЛА is
     record
     ВНУТРЕННЕЕ_ИМЯ: INTEGER: =0;
     end record,
end ПАКЕТ_ВВОДА_ВЫВОДА;
package body ПАКЕТ_ВВОДА_ВЫВОДА is
  ПРЕДЕЛ: constant: = 200;
  type ДЕСКРИПТОР_ФАЙЛА is record . . . end record;
  СПРАВОЧНИК: array (1. .ПРЕДЕЛ) of ДЕСКРИПТОР_ФАЙЛА;
  . . .
  procedure ОТКРЫТЬ (F: in out ИМЯ –ФАЙЛА) is . . . end;
  procedure ЗАКРЫТЬ (F: in out ИМЯ_ФАЙЛА) is . . . end;
  procedure ЧИТАТЬ (F: in ИМЯ_ФАЙЛА; ЭЛЕМЕНТ: out INTEGER) is . . . end;
  procedure ПИСАТЬ (F; in ИМЯ_ФАЙЛА; ЭЛЕМЕНТ: in INTEGER) is . . . end;
begin
  . . .
end ПАКЕТ_ВВОДА_ВЫВОДА;
```

Примечание к примеру. В приведенном примере для внешних подпрограмм, использующих ПАКЕТ_ВВОДА_ВЫВОДА, имя файла можно получить в результате вызова процедуры ОТКРЫТЬ, а затем использовать его в вызовах процедур ЧИТАТЬ и ПИСАТЬ. Следовательно, вне пакета имя файла, полученное после вызова процедуры

ОТКРЫТЬ, выполняет функцию пароля; его внутренние свойства (например, содержать числовое значение) неизвестны и никакие другие операции (такие как сложение или сравнение внутренних имен) над этим именем не могут выполняться.

Этот пример характерен для тех случаев, когда желателен полный контроль над операциями. Такие пакеты служат двум целям: они препятствуют пользователю в использовании внутренней структуры типа; а также реализуют понятие упрятывания (скрытия) типа данных, для которого определены только заданные в спецификации пакета операции.

7.5. **Пример пакета работы с таблицами**

Следующий пример иллюстрирует использование пакетов для организации простого взаимодействия пользователя с довольно сложными процедурами.

Необходимо создать пакет для работы с таблицами по внесению и извлечению их элементов. Элементы включаются в таблицу по мере их поступления. Каждый поступивший элемент имеет порядковый номер. Элементы выбираются в соответствии с их порядковыми номерами, причем первым выбирается элемент с наименьшим порядковым номером.

С точки зрения пользователя пакет чрезвычайно прост. Существует тип с именем ЭЛЕМЕНТ – тип элемента таблицы, есть процедура ВНЕСТИ для включения элементов в таблицу и процедура ВЫБРАТЬ для извлечения элемента с наименьшим порядковым номером. Если таблица пуста, то возвращается специальный элемент НУЛЕВОЙ_ЭЛЕМЕНТ, а если таблица заполнена, то при вызове процедуры ВНЕСТИ возбуждается исключение ТАБЛИЦА_ПЕРЕПОЛНЕНА.

Ниже приведена схема такого пакета. Пользователю известна только спецификация пакета.

```
package УПРАВЛЕНИЕ_ТАБЛИЦЕЙ is
  type ЭЛЕМЕНТ is
     record
        ПОРЯДКОВЫЙ_НОМЕР: INTEGER;
        КОД_ЭЛЕМЕНТА: INTEGER;
        КОЛИЧЕСТВО: INTEGER;
        ТИП_ЭЛЕМЕНТА: CHARACTER;
     end record;
  НУЛЕВОЙ_ЭЛЕМЕНТ: constant ЭЛЕМЕНТ: =
     (ПОРЯДКОВЫЙ_НОМЕР | КОД_ЭЛЕМЕНТА | КОЛИЧЕСТВО => 0,
     ТИП_ЭЛЕМЕНТА => ' ');
  procedure ВНЕСТИ (НОВЫЙ_ЭЛЕМЕНТ: in ЭЛЕМЕНТ);
  procedure ВЫБРАТЬ (ПЕРВЫЙ_ЭЛЕМЕНТ: out ЭЛЕМЕНТ);
  ТАБЛИЦА_ПЕРЕПОЛНЕНА: exception;
  - - это исключение возбуждается в процедуре ВНЕСТИ, если таблица заполнена
end;
```

Детали реализации таких пакетов могут быть достаточно сложными; в данном случае используются двусвязные списки внутренних элементов. Локальная вспомогательная процедура ИЗМЕНИТЬ используется для перемещения внутренних элементов из списка занятых в список свободных. Начальные связи таблицы устанавливаются в разделе инициализации. Нет необходимости показывать пользователям тело пакета.

```
package body УПРАВЛЕНИЕ_ТАБЛИЦЕЙ is
  РАЗМЕР: constant: = 2000;
```

```
  subtype ИНДЕКС is INTEGER range 0. .РАЗМЕР;
  type ВНУТРЕННИЙ_ЭЛЕМЕНТ is
  record
     СОДЕРЖИМОЕ: ЭЛЕМЕНТ;
     ПОСЛЕДУЮЩИЙ; ИНДЕКС;
     ПРЕДЫДУЩИЙ: ИНДЕКС;
  end record;
ТАБЛИЦА: array (ИНДЕКС) of ВНУТРЕННИЙ_ЭЛЕМЕНТ;
ПЕРВЫЙ_ЗАНЯТЫЙ_ЭЛЕМЕНТ: ИНДЕКС: = 0;
ПЕРВЫЙ_СВОБОДНЫЙ_ЭЛЕМЕНТ: ИНДЕКС: = 1;
function СПИСОК_СВОБОДНЫХ_ЭЛЕМЕНТОВ_ПУСТ return BOOLEAN is. . .end;
function СПИСОК_ЗАНЯТЫХ_ЭЛЕМЕНТОВ_ПУСТ return BOOLEAN is. . .end;
procedure ИЗМЕНИТЬ (ИЗ: in ИНДЕКС; К: in ИНДЕКС) is . . . end;
procedure ВНЕСТИ (НОВЫЙ_ЭЛЕМЕНТ: in ЭЛЕМЕНТ) is
begin
  if СПИСОК_СВОБОДНЫХ_ЭЛЕМЕНТОВ_ПУСТ then
     raise ТАБЛИЦА_ПЕРЕПОЛНЕНА;
  end if;
  - - остальная часть кода подпрограммы ВНЕСТИ
end ВНЕСТИ;
procedure ВЫБРАТЬ (ПЕРВЫЙ_ЭЛЕМЕНТ: out ЭЛЕМЕНТ) is . . . end;
begin
  - - инициализация связей таблицы
end УПРАВЛЕНИЕ_ТАБЛИЦЕЙ;
```

7.6. **Пример пакета обработки текстов**

Этот пример иллюстрирует простой пакет обработки текстов. Пользователи имеют доступ только к видимому разделу; реализация от них скрыта в личном разделе и в теле пакета (тело не показано).

С точки зрения пользователя ТЕКСТ является строкой переменной длины. Каждый текстовый объект имеет максимальную длину, которая должна задаваться при описании этого объекта, и текущую длину, которая равна длине в диапазоне от нуля до максимального. Максимальная возможная длина текстового объекта является константой, определяемой реализацией.

Сначала в пакете определяются необходимые типы, затем функции, возвращающие некоторые характеристики объектов типа, затем функции преобразования текстов и предопределенных типов CHARACTER и STRING и, наконец, некоторые стандартные операции над переменными строками. Большинство операций над строками, символами, а также над типом ТЕКСТ, совмещены для минимизации числа явных преобразований, которые должен написать пользователь.

```
package ОБРАБОТКА_ТЕКСТА is
  МАКСИМУМ: constant: = НЕКОТОРОЕ_ЗНАЧЕНИЕ;
  - - это значение определено реализацией
  subtype ИНДЕКС is INTEGER range 0. .МАКСИМУМ;
  type ТЕКСТ (МАКСИМАЛЬНАЯ_ДЛИНА: ИНДЕКС) is limited private;
  function ДЛИНА (Т: ТЕКСТ) return ИНДЕКС;
  function ЗНАЧЕНИЕ (Т: ТЕКСТ) return STRING;
  function ПУСТО (Т: ТЕКСТ) return BOOLEAN;
  function К_ТЕКСТУ (S: STRING; МАКС: ИНДЕКС) return ТЕКСТ;
  - - МАКС – максимальная длина
  function К_ТЕКСТУ (С: CHARACTER; МАКС: ИНДЕКС) return ТЕКСТ;
```

```
    function К_ТЕКСТУ (S: STRING) return ТЕКСТ;
    -- S'LENGTH – максимальная длина
    function К_ТЕКСТУ (С: CHARACTER) return ТЕКСТ;
    function "&" (СЛЕВА: ТЕКСТ; СПРАВА: ТЕКСТ) return ТЕКСТ;
    function "&" (СЛЕВА: ТЕКСТ; СПРАВА: STRING) return ТЕКСТ;
    function "&" (СЛЕВА: STRING; СПРАВА: ТЕКСТ) return ТЕКСТ;
    function "&" (СЛЕВА: ТЕКСТ; СПРАВА: CHARACTER) return ТЕКСТ;
    function "&" (СЛЕВА: CHARACTER; СПРАВА: ТЕКСТ) return ТЕКСТ;
    function "=" (СЛЕВА: ТЕКСТ; СПРАВА: ТЕКСТ) return BOOLEAN;
    function "<" (СЛЕВА: ТЕКСТ; СПРАВА: ТЕКСТ) return BOOLEAN;
    function "<=" (СЛЕВА: ТЕКСТ; СПРАВА: ТЕКСТ) return BOOLEAN;
    function ">" (СЛЕВА: ТЕКСТ; СПРАВА: ТЕКСТ) return BOOLEAN;
    function ">=" (СЛЕВА: ТЕКСТ; СПРАВА: ТЕКСТ) return BOOLEAN;
    procedure ПОМЕСТИТЬ (ОБЪЕКТ: in out ТЕКСТ; ЗНАЧЕНИЕ: in ТЕКСТ);
    procedure ПОМЕСТИТЬ (ОБЪЕКТ: in out ТЕКСТ; ЗНАЧЕНИЕ: in STRING);
    procedure ПОМЕСТИТЬ (ОБЪЕКТ: in out ТЕКСТ; ЗНАЧЕНИЕ: in CHARACTER);
    procedure СОЕДИНИТЬ (ЧАСТЬ: in ТЕКСТ; К: in out ТЕКСТ);
    procedure СОЕДИНИТЬ (ЧАСТЬ: in STRING; К: in out ТЕКСТ);
    procedure СОЕДИНИТЬ (ЧАСТЬ: in CHARACTER; К: in out ТЕКСТ);
    procedure ЗАМЕНИТЬ (ОБЪЕКТ: in out ТЕКСТ; НА: in ТЕКСТ;
                        ПОЗИЦИЯ: in ИНДЕКС);
    procedure ЗАМЕНИТЬ (ОБЪЕКТ: in out ТЕКСТ; НА: in STRING;
                        ПОЗИЦИЯ: in ИНДЕКС);
    procedure ЗАМЕНИТЬ (ОБЪЕКТ: in out ТЕКСТ; НА in CHARACTER;
                        ПОЗИЦИЯ: in ИНДЕКС);
    -- заменяет часть объекта с заданной позиции на данный
    -- текст, строку или символ
    function ВСТАВИТЬ (ФРАГМЕНТ: ТЕКСТ; В: ТЕКСТ) return ИНДЕКС;
    function ВСТАВИТЬ (ФРАГМЕНТ: STRING; В: ТЕКСТ) return ИНДЕКС;
    function ВСТАВИТЬ (ФРАГМЕНТ: CHARACTER; В: ТЕКСТ) return ИНДЕКС;
    -- возвращают значение 0, если фрагмент не помещается
  private
    type ТЕКСТ (МАКСИМАЛЬНАЯ_ДЛИНА: ИНДЕКС) is
      record
        ПОЗИЦИЯ: ИНДЕКС: = 0;
        ЗНАЧЕНИЕ: STRING (1..МАКСИМАЛЬНАЯ_ДЛИНА);
      end record;
  end ОБРАБОТКА_ТЕКСТА;
```

Пример использования пакета обработки текста:

Программа открывает файл вывода, имя которого дается строкой ИМЯ. Эта строка имеет вид:

[УСТРОЙСТВО:] [ИМЯФАЙЛА [. РАСШИРЕНИЕ]]

Для устройства имени файла и расширения существуют стандартные значения по умолчанию. Названное пользователем имя передается через параметр функции РАСШИРЕННОЕ_ИМЯ_ФАЙЛА, ее результатом является расширенная версия с необходимыми добавлениями по умолчанию.

```
  function РАСШИРЕННОЕ_ИМЯ_ФАЙЛА (ИМЯ: STRING) return STRING is
    use ОБРАБОТКА_ТЕКСТА;
    УСТРОЙСТВО_ПО_УМОЛЧАНИЮ: constant STRING: = "SY:";
    ИМЯ_ФАЙЛА_ПО_УМОЛЧАНИЮ: constant STRING: = "RESULTS";
    РАСШИРЕННОЕ_ПО_УМОЛЧАНИЮ: constant STRING: = "DAT";
    МАКСИМАЛЬНАЯ_ДЛИНА_ИМЕНИ_ФАЙЛА: constant ИНДЕКС: =
          НЕКОТОРОЕ_ПОДХОДЯЩЕЕ_ЗНАЧЕНИЕ;
    ИМЯ_ФАЙЛА: ТЕКСТ (МАКСИМАЛЬНАЯ_ДЛИНА_ИМЕНИ_ФАЙЛА);
```

```
begin
  ПОМЕСТИТЬ (ИМЯ_ФАЙЛА, ИМЯ);
  if ПУСТО (ИМЯ_ФАЙЛА) then
     ПОМЕСТИТЬ (ИМЯ_ФАЙЛА, ИМЯ_ФАЙЛА_ПО_УМОЛЧАНИЮ);
  end if;
  if ВСТАВИТЬ (':', ИМЯ_ФАЙЛА) = 0 then
     ПОМЕСТИТЬ (ИМЯ_ФАЙЛА, УСТРОЙСТВО_ПО_УМОЛЧАНИЮ &
                ИМЯ_ФАЙЛА);
  end if;
  if ВСТАВИТЬ ('.', ИМЯ_ФАЙЛА) = 0 then
     СОЕДИНИТЬ (РАСШИРЕНИЕ_ПО_УМОЛЧАНИЮ, К => ИМЯ_ФАЙЛА);
  end if;
  return ЗНАЧЕНИЕ (ИМЯ_ФАЙЛА);
end РАСШИРЕННОЕ_ИМЯ_ФАЙЛА;
```

8. ПРАВИЛА ВИДИМОСТИ

В этой главе описаны правила, определяющие область действия описания, и правила, определяющие видимость идентификаторов в различных точках текста программы. Формулировка правил видимости использует понятие зоны описания.

8.1. **Зона описания**

Зона описания является частью текста программы. Отдельная зона описания – это:

• Описание подпрограммы, описание пакета, описание задачи или описание настройки с соответствующим телом (если оно есть). Если это тело – след тела, то зона описания включает также соответствующий субмодуль. Если программный модуль содержит субмодули, то они также включаются в зону описания.

• Описание входа с соответствующими операторами принятия.

• Описание именуемого типа с соответствующими описанием личного типа или неполным описанием типа (если они есть) и спецификатором представления записи (если он есть).

• Описание переименования, содержащее раздел формальных параметров, или описание параметров настройки, включающее либо раздел формальных параметров, либо раздел дискриминантов.

• Оператор блока или оператор цикла.

В каждом из перечисленных выше случаев говорят, что зона описания *связана* с соответствующим описанием или оператором. Описание *находится непосредственно* в зоне описания, если она является самой вложенной, охватывающей данное описание, без учета зоны описания (если она есть), связанной с самим описанием.

Описание, которое находится непосредственно в зоне описания, является *локальным* в этой зоне. Описания во внешних (охватывающих) зонах называются *глобальными* по отношению к внутренней (охватываемой) зоне описания. Локальные понятия – это те, которые описаны непосредственно локальными описаниями; глобальные понятия – это те, которые описаны посредством глобальных описаний.

Некоторые из упомянутых выше форм зон описания включают несколько разъединенных разделов (например, между описанием пакета и его телом могут быть помещены другие элементы описания). Тем не менее, каждая зона описания рассматривается как непрерывная часть текста программы (логически). Следовательно, если какое-либо правило определяет часть текста, *расположенного* от некоторой выделенной точки зоны описания до конца зоны, то эта часть является соответствующим подмножеством данной зоны описания (в нее не включаются, например, промежуточные элементы описания, расположенные между двумя разделами пакета).

Примечание. Как определено в разд. 3.1, в термин описание включаются основные описания, неявные описания и описания, являющиеся разделом основных описаний, например, спецификации дискриминантов и параметров. Из определения зоны описания следует, что спецификация дискриминанта находится непосредственно в зоне, связанной с охватывающим описанием именуемого типа. Аналогично, спецификация параметра находится непосредственно в зоне, связанной с телом охватывающей подпрограммы или оператором принятия.

Пакет STANDARD образует зону описания, которая охватывает все библиотечные модули; предполагается, что неявное описание каждого библиотечного модуля находится непосредственно в этом пакете (см. разд. 8.6 и 10.1.1).

Зоны описания могут быть вложены в другие зоны описания. Например, подпрограммы, пакеты, задачные модули, настраиваемые модули и операторы блока могут быть вложены друг в друга и содержать описания именуемого типа, оператор цикла, а также операторы принятия.

8.2. Области действия описаний

Для каждой формы описания правила языка определяют конкретную часть текста программы, называемую *областью действия* описания или областью действия описанного понятия. Более того, если описание сопоставляет некоторое обозначение с описанным понятием, то эта часть текста также называется областью действия этого обозначения (либо идентификатора, либо символьного литерала, либо знака операции, либо обозначения базовой операции). В области действия понятия, и только в ней, есть места, в *которых будет правильным использовать сопоставленное обозначение для* ссылки на описанное понятие. Эти места определены правилами видимости и совмещения.

Область действия описания, находящегося непосредственно в зоне описания, распространяется от начала описания до конца зоны описания; этот раздел области действия описания называется *непосредственной областью действия*. Более того, для любого из описаний, перечисленных ниже, область действия описаний распространяется за пределы непосредственной области действия:

а) описание, которое находится непосредственно в видимом разделе описания пакета;

б) описание входа;

в) описание компонента;

г) спецификация дискриминанта;

д) спецификация параметра;

е) описание параметра настройки.

В каждом из этих случаев данное описание находится непосредственно в некотором охватывающем описании, а область действия данного описания распространяется до конца области действия охватывающего описания.

При отсутствии описания подпрограммы спецификация подпрограммы, заданная в теле подпрограммы или в следе тела, действует как описание, и в этом случае применимо правило *д*.

Примечание. Приведенные правила, определяющие область действия, применяются для всех форм описаний, определенных в разд. 3.1; они применяются, в частности, и к неявным описаниям. Правило *а* применяется к описанию пакета и тем самым неприменимо к спецификации пакета в описании настройки. Для вложенных описаний правила от *а* до *е* применяются на каждом уровне. Например, если задачный модуль описан в видимом разделе пакета, то область действия входа задачного модуля распространяется до конца области действия этого задачного модуля, т. е. до конца области действия охватывающего пакета. Область действия спецификатора использования определена в разд. 8.4.

8.3. **Видимость**

Правила видимости, а в случае совмещенных описаний и правила совмещения трактуют вхождение идентификатора в данной точке текста программы. Под идентификатором в данной главе подразумевают любой идентификатор, кроме зарезервированных слов, обозначений атрибутов, идентификаторов прагм и аргументов прагм. Под точкой текста программы в этой главе понимают место вхождения лексемы (например, идентификатора), а под совмещенными описаниями – описания подпрограмм, литералов перечисления, одиночных входов.

Для каждого идентификатора и в каждой точке текста программы правила видимости определяют набор описаний (с этим идентификатором), т. е. варианты трактовки идентификатора. Говорят, что описание *видимо* в данной точке текста, когда, согласно правилам видимости, оно определяет возможные трактовки его вхождения. Возникают два случая:

• Правила видимости определяют *не более одной* трактовки идентификатора. В таком случае правил видимости достаточно для выявления описания, определяющего трактовку вхождения идентификатора, или, при отсутствии такого описания, для выявления того, что это вхождение в данной точке не является правильным.

• Правила видимости определяют *более чем одну* трактовку. В таком случае вхождение идентификатора является правильным в данной точке, если и только если *точно одно* видимое описание выбирается правилами совмещения в соответствии с данным контекстом (см. разд. 6.6 для правил совмещения и разд. 8.7 для контекста, используемого при разрешении совмещения).

Описание видимо только в определенной части своей области действия; эта часть начинается в конце описания, а в спецификации пакета эта часть начинается с зарезервированного слова **is**, следующего за идентификатором пакета. (Это правило применяется, в частности, для неявных описаний.)

Видимость может быть прямой или видимостью по имени. Описание видимо *по имени* в местах, которые определены следующим образом:

а) для описания, находящегося в видимом разделе описания пакета, – на месте постфикса после точки в расширенном имени, префикс которого обозначает пакет;

б) для описания входа конкретного задачного типа – на месте постфикса после точки в именованном компоненте, префикс которого соответствует задачному типу;

в) для описания компонента конкретного описания именуемого типа – на месте постфикса после точки в именованном компоненте, префикс которого соответствует этому типу, а также на месте простого имени компонента (перед составным ограничителем =>) в именованном сопоставлении компонентов агрегата этого типа;

г) для спецификации дискриминанта конкретного описания типа – в местах, предназначенных для описания компонента и простого имени дискриминанта (перед составным ограничителем =>) в именованном сопоставлении дискриминанта в ограничении дискриминанта для этого типа;

д) для спецификации параметра данной спецификации подпрограммы или описания входа – на месте формального параметра (перед составным ограничителем =>) в именованном сопоставлении параметра соответствующей подпрограммы или вызова входа;

е) для описания параметра настройки данного настраиваемого модуля – на месте формального параметра настройки (перед составным ограничителем =>) в именованном сопоставлении соответствующей конкретизации настройки.

Наконец, в зоне описания, связанной с конструкцией, не являющейся описанием именуемого типа, любое описание видимо по имени на месте постфикса после точки в расширенном имени, префикс которого обозначает эту конструкцию.

Там, где нет видимости по имени, говорят, что видимое описание *видимо непосредственно* (прямо видимо). Описание видимо непосредственно в определенном разделе его непосредственной области действия; этот раздел распространяется до конца непосредственной области действия описания, за исключением тех мест, где это описание скрыто, как поясняется ниже. Кроме того, описание, находящееся непосредственно в видимом разделе пакета, может быть сделано непосредственно видимым с помощью спецификатора использования по правилам, описанным в разд. 8.4 (см. также разд. 8.6 о видимости библиотечных модулей).

Описание *скрыто* во внутренней зоне описания, если она содержит омограф этого описания; внешнее описание является тогда скрытым в непосредственной области действия этого внутреннего омографа. Каждое из двух описаний является *омографом* другого, если оба описания имеют один и тот же идентификатор и не более чем для одного из них допустимо совмещение. Если совмещение допустимо для обоих описаний, то каждое из двух является омографом другого, в случае если они имеют одинаковый

идентификатор, символ операции или символьный литерал, а также одинаковый профиль типа параметров и результата (см. разд. 6.6).

В спецификации подпрограммы скрыто каждое описание с таким же обозначением, как у подпрограммы; данное положение справедливо для конкретизации настройки, которая описывает подпрограмму, и в описании входа или в разделе формальных параметров оператора принятия. В этих случаях описание не является видимым ни по имени, ни непосредственно.

Два описания, которые находятся непосредственно в одной и той же зоне описания, не должны быть омографами, за исключением тех случаев, когда выполнены одно или оба следующих требования:

а) только одно из них является неявным описанием предопределенной операции;

б) только одно из них является неявным описанием производной подпрограммы.

В таких случаях предопределенная операция всегда скрыта другим омографом производная подпрограмма скрывает предопределенную операцию, но скрыта сама любым другим омографом. Там, где скрытие осуществляется таким образом, неявное описание скрыто во всей области действия другого описания (независимо от того, какое описание стоит первым); неявное описание не видимо ни по имени, ни непосредственно.

Всегда, когда описание с определенным идентификатором видимо в данной точке, говорят, что идентификатор и описанное понятие (если оно есть) видимы в этой точке. Непосредственная видимость и видимость по имени для символьных литералов и знаков операции определяется аналогично. Операция, обозначенная знаком, непосредственно видима тогда и только тогда, когда описание соответствующей операции непосредственно видимо. Наконец, обозначения, связанные с базовой операцией, непосредственно видимы во всей области действия этой операции.

Пример:

```
procedure P is
  A, B: BOOLEAN;
  procedure Q is
     C: BOOLEAN;
     B: BOOLEAN; - - внутренний омограф B
  begin
  ...
     B: = A; - - означает Q.B: = P.A;
     C: = P.B; - - означает Q.C: = P.B;
  end;
begin
  ...
     A: = B; - - означает P.A: = P.B;
  end;
```

Примечание о видимости библиотечных модулей. Видимость библиотечных модулей определена спецификаторами совместности (см. разд. 10.1.1) и тем фактом, что библиотечные модули неявно описаны в пакете STANDARD (см. разд. 8.6).

Примечание об омографах. Один и тот же идентификатор может находиться в различных описаниях и, таким образом, может соответствовать различным понятиям, да-

же если области действия описаний перекрываются. Перекрытие областей действия описаний с одним и тем же идентификатором может получиться из-за совмещения подпрограмм и литералов перечисления. Такое перекрытие может также произойти для понятий, описанных в видимых разделах пакета, а также входов, компонентов записей и параметров, где имеется перекрытие областей действия охватывающих описаний пакета, описаний задачи, описаний именуемого типа, описаний подпрограмм, описаний переименований и описаний настройки. Наконец, перекрытие областей действия может быть результатом вложенности.

Примечание к непосредственной области действия, скрытию и видимости. Правила, определяющие непосредственную область действия, скрытия и видимости, предусматривают, что ссылка на идентификатор в его собственном описании является неправильной (исключая случаи пакетов и настраиваемых пакетов). Идентификатор скрывает внешние омографы в собственной непосредственной области действия (от начала описаний в области); с другой стороны, идентификатор является видимым только после конца описания. По этой причине все следующие описания (кроме последнего) являются неправильными:

```
K: INTEGER: = K * K;                    - - неправильно
T : T;                                  - - неправильно
procedure P (X:P) ;                     - - неправильно
procedure Q (X: ВЕЩЕСТВ : = Q);         - - неправильно
        - - даже если существует функция с именем Q
procedure R (R: ВЕЩЕСТВ) ;  - - внутреннее описание, правильное, хотя создает
                            - - путаницу
```

8.4. **Спецификаторы использования**

Спецификатор использования обеспечивает непосредственную видимость описаний, которые находятся в видимых разделах пакетов с именами, упомянутыми в спецификаторе использования.

спецификатор_использования : : = use имя_*пакета* {, имя_*пакета*};

Для каждого спецификатора использования существует определенная зона текста, называемая *областью действия* спецификатора использования. Эта зона начинается непосредственно после спецификатора использования. Если спецификатор использования является элементом описания некоторой зоны описания, то область действия спецификатора использования распространяется до конца этой зоны описания. Если спецификатор использования находится в спецификаторе контекста компилируемого модуля, то область действия спецификатора использования распространяется до конца зоны описания, связанной с данным компилируемым модулем.

Чтобы определить, какие описания становятся прямо видимыми в данном месте с помощью спецификаторов использования, рассмотрим все пакеты, упомянутые в спецификаторах использования, области действия которых (спецификаторов) охватывают это место. Описанием, которое может быть сделано прямо видимым с помощью спецификатора использования (*потенциально видимое описание*), является такое описание, которое находится непосредственно в видимом разделе одного из этих пакетов. Потенциально видимое описание становится фактически прямо видимым, за исключением двух случаев:

• Потенциально видимое описание не становится прямо видимым, если рассматриваемое место программы находится в непосредственной области действия описания омографа.

• Потенциально видимые описания с одинаковыми идентификаторами не становятся прямо видимыми, если только каждое из них не является спецификацией литерала перечисления или описанием подпрограммы (представляющим собой описание подпрограммы, описание переименования, конкретизацию настройки или неявное описание).

Предвыполнение спецификатора использования не имеет другого эффекта.

Примечание. Приведенные выше правила гарантируют, что описание, которое стало прямо видимым с помощью спецификатора использования, не может скрывать другое прямо видимое описание. Эти правила сформулированы в терминах набора пакетов, упомянутых в спецификаторах использования.

Следовательно, приведенные ниже строчки текста дают один и тот же эффект (в предположении существования единственного пакета Р).

```
use P;
use P; use P, P;
```

Пример противоречия имен в двух пакетах:

```
procedure R is
  package ТРАФИК is
     type ЦВЕТ  is (КРАСНЫЙ, ЯНТАРНЫЙ, ЗЕЛЕНЫЙ);
     . . .
  end ТРАФИК;
  package АКВАРЕЛИ is
     type ЦВЕТ is (БЕЛЫЙ, КРАСНЫЙ, ЖЕЛТЫЙ, ЗЕЛЕНЫЙ, СИНИЙ,
                   КОРИЧНЕВЫЙ, ЧЕРНЫЙ);
     . . .
  end АКВАРЕЛИ;
  use ТРАФИК; - - ЦВЕТ, КРАСНЫЙ, ЯНТАРНЫЙ и ЗЕЛЕНЫЙ непосредственно
              - - видимы
  use АКВАРЕЛИ; - - два омографа ЗЕЛЕНЫЙ непосредственно
                - - видимы, но ЦВЕТ не является более
                - - непосредственно видимым
  subtype СВЕТ is ТРАФИК.ЦВЕТ; - - подтип использован
  - - для разрешения противоречия, связанного с именем типа ЦВЕТ;
  subtype ТЕНЬ is АКВАРЕЛИ.ЦВЕТ;
  СИГНАЛ: СВЕТ;
  КРАСКА : ТЕНЬ;
begin
  СИГНАЛ: = ЗЕЛЕНЫЙ; - - из пакета ТРАФИК
  КРАСКА: = ЗЕЛЕНЫЙ; - - из пакета АКВАРЕЛИ
end P;
```

Пример идентификации имени со спецификатором использования:

```
package Д is
  T, A, B: BOOLEAN;
end Д;
procedure P is
   package E is
     Б, C, B: INTEGER;
   end E;
   procedure H is
     T, X: ВЕЩЕСТВ;
     use Д, E;
```

```
begin
 -- имя Т означает Н.Т, а не Д.Т
 -- имя А означает Д.А
 -- имя Б означает Е.Б
 -- имя С означает Е.С
 -- имя Х означает Н.Х
 -- имя В неправильно: должно быть использовано Д.В или Е.В
    ...
 end H;
begin
    ...
end P;
```

8.5. **Описание переименования**

Описание переименования задает другое имя для понятия.

описание_переименования : : =

идентификатор : обозначение_типа **renames** имя_*объекта;*

| идентификатор : **exception renames** имя_*исключения;*

| **package** идентификатор **renames** имя_*пакета;*

| спецификация_подпрограммы **renames**

имя_*подпрограммы*_или_*входа*;

Предвыполнение описания переименования вычисляет имя, которое следует за резервированным словом **renames**, и, таким образом, определяет понятие, обозначенное этим именем (переименованное понятие). В любой точке, где описание переименования видимо, идентификатор или знак операции, заданный в этом описании, обозначает переименованное понятие.

Первая форма описания переименования используется для переименования объектов. Переименованное понятие должно быть объектом базового типа обозначения типа. Описание переименования не изменяет свойства переименованного объекта. В частности, описание переименования не оказывает влияния на значение объекта и на то, является ли он константой или нет; аналогично, переименование не затрагивает ограничения, накладываемые на объект (любое ограничение, которое следует из обозначения типа, входящего в описание переименования, игнорируется). Описание переименования правильно только в том случае, если точно один объект имеет этот тип и может быть обозначен этим именем объекта.

Существуют следующие ограничения, связанные с переименованием подкомпонента переменной, которая зависит от дискриминантов. Переименование недопустимо, если подтип переменной, как это определено в соответствующем описании объекта, описании компонента или указании подтипа компонента, является неограниченным типом или, если переменная – это формальный объект настройки (вида **in out**). Также, если переменная – формальный параметр, то переименование недопустимо, если заданное в спецификации параметра обозначение типа обозначает неограниченный тип, чьи дискриминанты имеют выражения по умолчанию.

Вторая форма описания переименования используется для переименования исключений; третья форма – для переименования пакетов.

Последняя форма описания переименования используется для переименования подпрограмм и входов. Переименованная подпрограмма или вход

и спецификация подпрограммы, заданная в описании переименования, должны иметь один и тот же профиль типа параметров и результата (см. разд. 6.6). Описание переименования правильно только в том случае, если точно одна видимая подпрограмма или вход удовлетворяют упомянутым выше требованиям и могут быть обозначены конкретным именем подпрограммы или входа. Кроме того, виды параметров должны совпадать с видами соответствующих по позиции формальных параметров.

Переименование не оказывает влияния на подтипы параметров и результата (если он есть) переименованной подпрограммы или входа. Эти подтипы заданы в первоначальном описании подпрограммы, конкретизации настройки или описании входа (но не в описании переименования), а также для вызовов, которые используют новое имя. С другой стороны, описание переименования может вводить имена параметров и выражения по умолчанию, которые отличаются от заданных для переименованной подпрограммы; именованные сопоставления в вызовах с новым именем подпрограммы должны использовать новое имя параметра; вызовы со старым именем подпрограммы должны использовать старые имена параметров.

Процедура может быть переименована только как процедура. Функция либо операция могут быть переименованы как функция либо как операция; при переименовании функции или операции операцией спецификация подпрограммы, заданная в описании переименования, подчиняется правилам разд. 6.7 для описаний операции. Литералы перечисления могут быть переименованы как функции; аналогично, атрибуты, определенные как функции (такие как SUCC или PRED), могут быть переименованы как функции. Вход может быть переименован только как процедура; новое имя допускается только в контексте, допускающем имя процедуры. Вход из семейства может быть переименован, но семейство входов не может быть переименовано целиком.

Примеры:

```
declare
  П: ПЕРСОНА renames КРАЙНЯЯ_ПЕРСОНА; - - см. 3.8.1
begin
  П. ВОЗРАСТ: = П.ВОЗРАСТ + 1;
end;
ПОЛНЫЙ: exception renames ТАБЛИЦА_УПРАВЛЕНИЯ. ТАБЛИЦА_ПОЛНОТЫ;
- - см. 7.5
package ТУ renames ТАБЛИЦА_УПРАВЛЕНИЯ;
function ВЕЩЕСТВ_ПЛЮС (ЛЕВЫЙ, ПРАВЫЙ: ВЕЩЕСТВ) return ВЕЩЕСТВ
                                renames "+";
function ЦЕЛ_ПЛЮС (ЛЕВЫЙ, ПРАВЫЙ: INTEGER) return INTEGER renames "+";
function КРАСН_ФРАН return ЦВЕТ renames КРАСНЫЙ; - - см. 3.5.1
function КРАС_НЕМ return ЦВЕТ renames КРАСНЫЙ;
function КРАСН_ИТАЛ return ЦВЕТ renames КРАСН_ФРАН;
function СЛЕДУЮЩИЙ (X: ЦВЕТ) return ЦВЕТ renames ЦВЕТ'SUCC; - - см. 3.5.5
```

Примеры описания переименования с новыми именами параметров:

```
function "*" (X, Y: ВЕКТОР) return ВЕЩЕСТВ renames СКАЛ_ПРОИЗВЕДЕНИЕ;
- - см. 6.1
```

Пример описания переименования с новым выражением по умолчанию:

function МИНИМУМ (А: СВЯЗЬ: = ГОЛОВА) **return** ЯЧЕЙКА **renames** МИН_ЯЧЕЙКА; - - см. 6.1

Примечание. Переименование может быть использовано для разрешения конфликта имен и для введения сокращений. Переименование другим идентификатором или символом операции не скрывает старое имя; новое и старое имя (символьная операция) не обязательно видимы в одних и тех же точках. Атрибуты POS и VAL не могут быть переименованы, так как не могут быть написаны соответствующие спецификации; это положение справедливо для предопределенных мультипликативных операций с результатом *универсального_фиксированного* типа.

Вызовы переименованного входа с новым именем являются операторами вызова процедуры и недопустимы в местах, где синтаксис требует оператора вызова входа в условном и временном вызовах входа; аналогично, атрибут COUNT нельзя применить к новому имени.

Объект задачного типа, описанный посредством описания объекта, может быть переименован как объект. Однако одиночная задача не может быть переименована, так как соответствующий задачный тип является анонимным. По тем же причинам не может быть переименован объект анонимного индексируемого типа. Не существует синтаксической формы для переименования настраиваемого модуля.

Для достижения эффекта переименования типа (включая задачный тип) может быть использован подтип, например:

subtype ВИД **is** TEXT_IO.FILE_MODE;

8.6. **Стандартный пакет**

Предопределенные типы (например, BOOLEAN, CHARACTER и INTEGER) описаны в предопределенном пакете, называемом STANDARD; этот пакет включает также описания предопределенных для них операций. Пакет STANDARD описан в обязательном приложении 3. Спецификация пакета STANDARD, за исключением предопределенных числовых типов, должна быть одинаковой для всех реализаций языка.

Пакет STANDARD образует зону описания, которая охватывает каждый библиотечный модуль и, следовательно, главную программу; предполагается, что описание каждого библиотечного модуля находится непосредственно в этом пакете. Предполагается также, что неявные описания библиотечных модулей упорядочены таким образом, что область действия данного библиотечного модуля включает в себя любой компилируемый модуль, который упоминает в спецификаторе совместности этот библиотечный модуль. Однако видимыми внутри данного компилируемого модуля являются только такие библиотечные модули, которые упомянуты во всех спецификаторах совместности, относящихся к данному модулю, и тот библиотечный модуль, по отношению к которому данный модуль является вторичным модулем.

Примечание. Если все вложенные операторы блока программы поименованы, то имя каждого программного модуля, вложенного в блок, всегда может быть записано как расширенное имя, начинающееся с идентификатора STANDARD (в случае, когда этот пакет не является скрытым).

Если тип описан в видимом разделе библиотечного пакета, то из правил видимости следует, что базовая операция (например, присваивание) над этим типом непосредственно видима в точке, где сам тип невидим (ни по имени, ни непосредственно). Однако эта операция может быть применена только к тем операндам, которые являются

видимыми, и описание этих операндов требует видимости либо типа, либо одного из его подтипов.

8.7. **Контекст разрешения совмещения**

Совмещение определено для подпрограмм, литералов перечисления, символов операций и одиночных входов, а также для тех операций, которые присущи обычным базовым операциям, например, присваивание, проверка принадлежности, генератор, литерал **null**, агрегаты и строковые литералы.

Для совмещенных понятий разрешение совмещения определяет фактический смысл, который имеет вхождение идентификатора, когда в соответствии с правилами видимости выясняется, что в месте этого вхождения приемлема более чем одна трактовка идентификатора; аналогичным образом разрешение совмещения определяет фактическую трактовку вхождения операции или некоторой базовой операции.

В таком месте рассматриваются все видимые описания. Вхождение правильно только тогда, когда есть точно одна интерпретация для каждого элемента самого вложенного полного контекста. Полный контекст – это:

- описание;
- оператор;
- спецификатор представления.

При рассмотрении возможных интерпретаций полного контекста учитываются только те правила, которые касаются синтаксиса, области действия и видимости, а также те, которые даны ниже. Учитываются:

а) любое правило, которое требует, чтобы имя или выражение имели определенный тип или такой же тип, как другое имя или выражение;

б) любое правило, которое требует, чтобы тип имени или выражения был типом определенного класса; аналогично, любое правило, которое требует, чтобы определенный тип был дискретным, целым, вещественным, универсальным, символьным, логическим или нелимитируемым типом;

в) любое правило, которое требует, чтобы префикс соответствовал определенному типу;

г) любое правило, которое задает определенный тип в качестве типа результата базовой операции, и любое правило, которое устанавливает, что это тип определенного класса;

д) правила, которые требуют, чтобы тип агрегата или строкового литерала был определен исключительно охватывающим полным контекстом (см. разд. 4.3 и 4.2). Аналогично, правила, которые требуют, чтобы тип префикса атрибута, тип выражения оператора выбора или тип операнда преобразования типа были определены независимо от контекста (см. разд. 4.1.4, 5.4, 4.6 и 6.4.1);

е) правила, данные в разд. 6.6 по разрешению вызовов совмещенных подпрограмм, в разд. 4.6 по неявным преобразованиям универсальных выражений, в разд. 3.6.1 по интерпретации дискретных диапазонов с границами, имеющими универсальный тип, в разд. 4.1.3 по интерпретации расширенного имени, чей префикс обозначает подпрограмму или оператор принятия.

Для имен подпрограмм, используемых в качестве аргументов прагмы, следуют другому правилу: прагма может применяться для нескольких совмещенных подпрограмм, как пояснено в разд. 6.3.2 для прагмы INLINE, в разд. 11.7 для прагмы SUPPRESS и в разд. 13.9 для прагмы INTERFACE.

Аналогично этому, данные в спецификаторах контекста (см. разд. 10.1.1) и спецификаторах адреса простые имена следуют другим правилам.

Примечание. Если существует только одна возможная интерпретация, то идентификатор обозначает соответствующее понятие. Однако данное утверждение не означает, что это вхождение обязательно правильно, так как существуют другие требования, которые не учитываются при разрешении совмещения; например, является ли выражение статическим, каковы виды параметров, является ли объект константой, выполняются ли правила согласования, является ли вхождение в спецификатор представления предписывающим, каков порядок предвыполнения и т. п.

Аналогично, при разрешении совмещения не учитываются подтипы. (Нарушение ограничения не делает программу неправильной, но возбуждает исключение во время выполнения программы).

Спецификация параметра цикла есть описание и, следовательно, полный контекст.

Правила, которые требуют, чтобы определенные конструкции имели один и тот же профиль параметров и типа результата, подпадают под категорию *а*; то же справедливо для правил, которые требуют согласования двух конструкций, так как это согласование требует в свою очередь, чтобы соответствующие имена имели одинаковый смысл, определенный правилами видимости и совмещения.

9. ЗАДАЧИ

Выполнение программы без задач определено в терминах последовательного выполнения ее действий в соответствии с правилами, сформулированными в других главах данного стандарта. Можно предположить, что эти действия выполняются одним *логическим процессором.*

Под *параллельным* выполнением задач понимают следующее. Предполагается, что каждую задачу выполняет отдельный логический процессор. Различные задачи (на различных логических процессорах) выполняются независимо, за исключением точек их синхронизации.

Некоторые задачи могут иметь *входы.* Вход задачи может быть *вызван* другими задачами. Задача *принимает* вызов одного из своих входов выполнением оператора принятия этого входа. Синхронизация достигается посредством *рандеву* между задачей, вызывающей вход, и задачей, принимающей вызов. Некоторые входы имеют параметры; вызовы входов и операторы принятия таких входов являются основным средством обмена значениями между задачами.

Свойства каждой задачи определяются соответствующим *задачным модулем,* который состоит из *спецификации задачи* и *тела задачи.* Задачные модули представляют собой одну из четырех форм программных модулей, из которых может состоять программа. Другие три формы – это подпрограммы, пакеты и настраиваемые модули. В данной главе описываются свойства задачных модулей, задач и входов и операторы, влияющие на взаимодействие задач (т. е. операторы вызова входов, операторы принятия, операторы задержки, операторы отбора и операторы прекращения).

Примечание. Параллельные задачи (параллельные логические процессоры) могут быть реализованы на многомашинных комплексах, многопроцессорных ЭВМ или чередующимся выполнением на одном *физическом процессоре.* С другой стороны, если реализация способна определить, что тот же результат получается при параллельном выполнении частей одной задачи на различных физических процессорах, то можно принять и такой способ выполнения; в этом случае несколько физических процессоров реализуют один логический процессор.

9.1. **Спецификации задач и тела задач**

Задачный модуль состоит из спецификации задачи и тела задачи. *Спецификация задачи,* которая начинается зарезервированными словами **task type**, описывает *задачный тип.* Значение объекта задачного типа указывает *задачу.* Если задача имеет входы, то они описываются в спецификации задачи; эти входы также называются входами объекта. Выполнение задачи определяется соответствующим телом задачи.

Спецификация задачи без зарезервированного слова **type** определяет одиночную *задачу.* Описание такой задачи эквивалентно описанию анонимного задачного типа одновременно с описанием объекта этого задачного типа, а идентификатор задачного модуля именует объект. В остальной части данной главы пояснения даются в терминах описаний задачного типа; соответствующие пояснения для одной задачи следуют из упомянутого отношения эквивалентности.

описание_задачи : : = спецификация_задачи;
спецификация_задачи : : =
task [**type**] идентификатор [**is**
{описание_входа}
{спецификатор_представления}
end [простое_имя_*задачи*]] ;
тело_задачи : : =
task body простое_имя_*задачи* **is**
[раздел_описаний]
begin
последовательность_операторов
[**exception**
обработчик_исключения
{обработчик_исключения}]
end [простое_имя_*задачи*] ;

Простое имя в начале тела задачи должно совпадать с идентификатором задачного модуля. Аналогично, если в конце спецификации или тела задачи появляется простое имя, то оно должно совпадать с идентификатором задачного модуля. Внутри тела задачи имя соответствующего задачного модуля может также быть использовано для ссылки на объект-задачу (указывать на задачу), тело которой выполняется в данный момент; кроме того, не допускается использование этого имени типа или производного от него типа или подтипа в качестве обозначения типа внутри собственно задачного модуля.

При предвыполнении спецификации задачи описания входов и спецификаторы представления (если они есть) предвыполняются в том порядке, в котором они даны. Спецификаторы представления применяются только ко входам, описанным в спецификации задачи (см. 13.5).

Предвыполнение тела задачи не имеет никакого другого результата, кроме установления, что тело с этих пор может быть использовано для выполнения задач, указанных объектами соответствующего задачного типа.

Выполнение тела задачи вызывается активизацией задачного объекта соответствующего типа (см. 9.3). Возможные в конце тела задачи обработчики исключений обрабатывают исключения, возбуждаемые в ходе выполнения последовательности операторов тела задачи (см. 11.4).

Примеры спецификации задачных типов:

```
task type РЕСУРС is
  entry ЗАХВАТИТЬ;
  entry ОСВОБОДИТЬ;
end РЕСУРС;
task type ДРАЙВЕР_КЛАВИАТУРЫ is
  entry ЧИТАТЬ (С: out CHARACTER);
  entry ПИСАТЬ (С: in CHARACTER);
end ДРАЙВЕР_КЛАВИАТУРЫ;
```

Примеры спецификации одной задачи:

```
task ПОСТАВЩИК_ПОТРЕБИТЕЛЬ is
  entry ЧИТАТЬ (V: out ЭЛЕМЕНТ);
  entry ПИСАТЬ (Е: in ЭЛЕМЕНТ);
end;
task КОНТРОЛЛЕР is
  entry ЗАПРОСИТЬ (УРОВЕНЬ) (D: ЭЛЕМЕНТ); - - семейство входов
end КОНТРОЛЛЕР;
task ПОЛЬЗОВАТЕЛЬ; - - не имеет входов
```

Пример спецификации задачи и соответствующего тела:

```
task ЗАЩИЩЕННЫЙ_МАССИВ is
- - ИНДЕКС и ЭЛЕМЕНТ – это глобальные типы
  entry ЧИТАТЬ (К: in ИНДЕКС; V: out ЭЛЕМЕНТ);
  entry ПИСАТЬ (К: in ИНДЕКС; Е; in ЭЛЕМЕНТ);
end;
task body ЗАЩИЩЕННЫЙ_МАССИВ is
  ТАБЛИЦА: array (ИНДЕКС) of ЭЛЕМЕНТ: = (ИНДЕКС => НУЛЬ_ЭЛЕМЕНТ);
begin
  loop
      select
        accept ЧИТАТЬ (К: in ИНДЕКС; V: out ЭЛЕМЕНТ) do
          V: = ТАБЛИЦА (К);
        end ЧИТАТЬ;
      or
        accept ПИСАТЬ (К: in ИНДЕКС; Е: in ЭЛЕМЕНТ) do
          ТАБЛИЦА (К): = Е;
        end ПИСАТЬ;
      end select;
  end loop;
end ЗАЩИЩЕННЫЙ_МАССИВ;
```

Примечание. Спецификация задачи задает интерфейс задачам данного типа с другими задачами тех же или различных типов, а также с главной программой.

9.2. **Задачные типы и задачные объекты**

Задачный тип является лимитируемым типом (см. 7.4.4). Следовательно, для объектов задачного типа не определены ни присваивание, ни предопределенное сравнение на равенство и неравенство; более того, вид **out** не допустим для формального параметра задачного типа.

Задачный объект – это объект задачного типа. Значение задачного объекта указывает задачу со входами соответствующего задачного типа, а ее выполнение определено соответствующим телом задачи. Если задачный объект является объектом или подкомпонентом объекта, заданными описанием объекта, то его значение определяется предвыполнением описания объекта. Если задачный объект является объектом или подкомпонентом объекта, созданными при выполнении генератора, то его значение определяется выполнением генератора. Для всех видов параметров, если фактический параметр указывает задачу, сопоставляемый формальный параметр указывает ту же задачу; это же относится к подкомпоненту фактического параметра и к соответствующему подкомпоненту сопоставляемого формального параметра; наконец, то же справедливо и для параметров настройки.

Примеры:

```
УПРАВЛЕНИЕ: РЕСУРС;
ТЕЛЕТАЙП: ДРАЙВЕР_КЛАВИАТУРЫ;
ПУЛ: array (1..10) of ДРАЙВЕР_КЛАВИАТУРЫ;
- - см. также примеры описаний одиночных задач в 9.1
```

Пример ссылочного типа, указывающего задачный объект:

```
type КЛАВИАТУРА is access ДРАЙВЕР_КЛАВИАТУРЫ;
ТЕРМИНАЛ: КЛАВИАТУРА: = new ДРАЙВЕР_КЛАВИАТУРЫ;
```

Примечание. Поскольку задачный тип является лимитируемым, он может появиться как определение лимитируемого личного типа в личном разделе и как фактический параметр настройки, сопоставляемый формальному параметру лимитируемого типа. С другой стороны, тип формального параметра настройки вида **in** не должен быть лимитируемым и, следовательно, не может быть задачным типом.

Задачные объекты ведут себя как константы (задачный объект всегда указывает одну и ту же задачу), поскольку их значения неявно определены либо при описании, либо при генерации, либо при сопоставлении параметров, и никакие присваивания недопустимы. Однако зарезервированное слово **constant** недопустимо в описании задачного объекта, так как его наличие требует явной инициализации. Задачный объект, который является формальным параметром вида **in**, есть константа (как и любой формальный параметр вида **in**).

Если алгоритм требует запоминания и переименования задачи, то это можно сделать определением ссылочного типа, указывающего на соответствующие задачные объекты, и использованием ссылочных значений для целей идентификации (см. предыдущий пример). Присваивание для такого ссылочного типа возможно, как и для любого другого ссылочного типа.

Для задачных типов допустимы описания подтипов, как и для других типов, но никакие ограничения к задачному типу не применимы.

9.3. **Выполнение и активизация задачи**

Тело задачи определяет выполнение всякой задачи, которая указывается задачным объектом соответствующего задачного типа. Начальный этап

этого выполнения называется *активизацией* задачного объекта и указанной им задачи; активизация состоит из предвыполнения раздела описаний (если он есть) тела задачи. Выполнение различных задач, в частности, их активизация, производится параллельно.

Если задачный объект описан непосредственно в разделе описаний, то активизация задачного объекта начинается после предвыполнения раздела описаний, предшествующего зарезервированному слову **begin**; аналогично, если такое описание помещено непосредственно в спецификацию пакета, то активизация начинается после предвыполнения раздела описаний тела пакета. То же относится и к активизации задачного объекта, являющегося подкомпонентом объекта, описанного непосредственно в разделе описаний или спецификации пакета. Первый оператор, следующий за разделом описаний, выполняется только после окончания активизации задачных объектов.

Если при активизации одной из таких задач возбуждается исключение, то эта задача становится законченной (см. 9.4); на других задачах это прямо не отражается. Если во время своей активизации одна из этих задач становится законченной, то после завершения (успешного или нет) активизации всех задач возбуждается исключение TASKING_ERROR; исключение возбуждается непосредственно за разделом описаний перед выполнением первого оператора (непосредственно после зарезервированного слова **begin**). Исключение TASKING_ERROR возбуждается лишь однажды, даже если во время активизации сразу несколько задач становятся законченными таким способом.

Если исключение возбуждается при предвыполнении раздела описаний или спецификации пакета, то любая созданная (прямо или косвенно) этим предвыполнением задача, которая еще не активизирована, становится завершенной, и, таким образом, она никогда не активизируется (см. разд. 9.4 с определением завершенной задачи).

Приведенные выше правила предполагают, что в теле пакета без операторов присутствует пустой оператор. Для пакета без тела подразумевается тело пакета с единственным пустым оператором. Если пакет без тела описывается непосредственно в некотором программном модуле или в операторе блока, то его тело подразумевается в конце раздела описаний программного модуля или оператора блока; при наличии нескольких пакетов без тела порядок следования подразумеваемых тел пакетов неопределен.

Задачный объект, являющийся объектом или подкомпонентом объекта, созданного выполнением генератора, активизируется этим выполнением. Активизация начинается после инициализации объекта, созданного генератором; если несколько подкомпонентов являются задачными объектами, они активизируются параллельно. Ссылочное значение, указывающее этот объект, возвращается генератором только после проведения этих активизаций.

Если исключение возбуждается при активизации одной из таких задач, то она становится законченной задачей; на другие задачи этот факт не оказывает прямого воздействия. Если во время своей активизации одна из

этих задач становится законченной таким образом, то после проведения (успешного или нет) активизации всех этих задач возбуждается исключение TASKING_ERROR; исключение возбуждается в той точке программы, где выполняется генератор. Исключение TASKING_ERROR возбуждается лишь однажды, даже если во время активизации сразу несколько задач становятся законченными таким образом.

Если исключение возбуждается во время инициализации объекта, созданного генератором (следовательно, до начала активизации), то любая задача, указанная подкомпонентом этого объекта, становится завершенной, и, таким образом, она никогда не активизируется.

Пример:

```
procedure P is
  A, B: РЕСУРС; -- предвыполняет задачные объекты A и B
  C: РЕСУРС; -- предвыполняет задачный объект C
begin
  -- задачи A, B, C активизируются параллельно
  -- перед выполнением первого оператора
...
end;
```

Примечание. Вход задачи может быть вызван до активизации задачи. Если несколько задач активизируются параллельно, выполнение любой из них не предполагает ожидания конца активизации других задач. Задача может стать законченной во время ее активизации как из-за исключения, так и из-за прекращения (см. 9.10).

9.4. Зависимость и завершение задач

Каждая задача *зависит* по крайней мере от одного родителя. *Родитель* – это конструкция, являющаяся либо задачей, либо в данный момент выполняемым оператором блока или подпрограммой, либо библиотечным пакетом (но не описанным в другом программном модуле). Зависимость от родителя является непосредственной зависимостью в следующих двух случаях:

а) Задача, указанная задачным объектом, который является объектом или подкомпонентом объекта, созданными при выполнении генератора, зависит от родителя, предвыполняющего соответствующее описание ссылочного типа.

б) Задача, указанная другим задачным объектом, зависит от родителя, выполнение которого создает задачный объект.

Более того, если задача зависит от данного родителя, являющегося оператором блока, выполняемым другим родителем, то задача также косвенно зависит и от этого родителя; то же справедливо, если данный родитель является подпрограммой, вызванной другим родителем, а также если данный родитель – задача, зависящая (прямо или косвенно) от другого родителя. Зависимости существуют и для объектов личного типа, полное описание которого задано в терминах задачного типа.

Говорят, что задача *закончила* свое выполнение, когда осуществилось выполнение последовательности операторов, помещенных в ее теле за резервированным словом **begin**. Аналогично этому, блок или подпрограмма

закончили свое выполнение, когда осуществилось выполнение соответствующей последовательности операторов. В случае оператора блока также говорят, что выполнение его закончилось при достижении операторов выхода, возврата или перехода, передающих управление из блока. В случае процедуры также говорят, что ее выполнение закончилось при достижении соответствующего оператора возврата. В случае функции также говорят, что ее выполнение закончилось после вычисления результирующего выражения в операторе возврата. Наконец, выполнение задачи, оператора блока или подпрограммы закончено, если при выполнении содержащихся в них соответствующих последовательностей операторов возбуждено исключение и нет соответствующего ему обработчика, а при его наличии — по окончании выполнения соответствующего обработчика.

Если у задачи нет зависимых задач и закончено ее выполнение, имеет место ее *завершение.* После завершения говорят, что она *завершена.* Если задача имеет зависимые задачи, то ее завершение имеет место после окончания выполнения задачи и завершения всех зависимых задач. Из оператора блока или тела подпрограммы, чье выполнение закончено, нельзя выйти до завершения всех зависимых задач.

С другой стороны, завершение задачи имеет место тогда и только тогда, когда ее выполнение достигло открытой альтернативы завершения в операторе отбора (см. 9.7.1) и удовлетворены следующие условия:

• Задача зависит от некоторого родителя, выполнение которого закончено (следовательно, не от библиотечного пакета).

• Каждая задача, зависящая от рассмотренного родителя, либо уже завершена, либо также ожидает открытой альтернативы завершения в операторе отбора.

Когда оба условия удовлетворены, задача становится завершенной вместе со всеми задачами, зависящими от этого же родителя.

Пример:

```
declare
  type ГЛОБАЛЬНЫЙ is access РЕСУРС; - - см. 9.1
  A, B: РЕСУРС;
  Г: ГЛОБАЛЬНЫЙ;
begin
  - - активизация A и B
  declare
      type ЛОКАЛЬНЫЙ is access РЕСУРС;
      X: ГЛОБАЛЬНЫЙ: = new РЕСУРС; - - активизация X. all
      Л: ЛОКАЛЬНЫЙ: = new РЕСУРС; - - активизация Л. all
      C: РЕСУРС;
  begin
      - - активизация C
      Г: = X; - - Г и X указывают один и тот же задачный объект
      . . .
  end; - - ожидание завершения C и Л. all, но не X. all
          . . .
end; - - ожидание завершения A, B и Г. all
```

Примечание. Правила завершения подразумевают, что все задачи, зависящие (прямо или косвенно) от данного родителя и еще не завершенные, могут завершиться

(коллективно) тогда и только тогда, когда каждая из них ожидает открытой альтернативы завершения в операторе отбора, и выполнение данного родителя закончено.

Те же правила справедливы и для главной программы. Следовательно, для завершения главной программы необходимо завершение всех зависимых задач, даже если соответствующий задачный тип описан в библиотечном пакете. С другой стороны, завершение главной программы не зависит от завершения задач, в свою очередь зависящих от библиотечных пакетов; в языке не определено, требуется ли завершение таких задач.

Для ссылочного типа, являющегося производным другого ссылочного типа, соответствующее определение ссылочного типа является определением родительского типа; зависимость в данном случае является зависимостью от родителя, который предвыполняет основные определения родительского ссылочного типа.

Описание переименования вводит новое имя для уже существующего понятия, и, следовательно, не порождает дальнейшей зависимости.

9.5. Входы, вызовы входов и операторы принятия

Вызовы входов и операторы принятия являются основными средствами синхронизации задач и передачи значений между задачами. Описание входа подобно описанию подпрограммы и допустимо только в спецификации задачи. Действия, которые следует выполнить после вызова входа, задаются соответствующими операторами принятия.

описание_входа : : =
  entry идентификатор [(дискретный диапазон)]
    [раздел_формальных_параметров]
оператор_вызова_входа : : =
  имя_*входа* [раздел_фактических_параметров] ;
оператор_принятия : : =
  accept простое_имя_*входа* [(индекс_входа)]
    [раздел_формальных_параметров] [**do**
      последовательность_операторов
  end [простое_имя_*входа*]] ;
индекс_входа : : = выражение

Описание входа, включающее дискретный диапазон (см. разд. 3.6.1), описывает *семейство* различных входов с одним и тем же разделом формальных параметров (если он есть), а именно по одному входу для каждого значения дискретного диапазона. Термин *одиночный вход* используется при определении правил, применимых к любому входу, отличному от члена семейства. Задача, указанная объектом задачного типа, имеет вход (входы), который (которые) описан (описаны) в спецификации этого задачного типа.

В теле задачи каждый из ее одиночных входов или семейства входов может быть именован соответствующим простым именем. Имя входа семейства записывается в форме индексируемого компонента: за простым именем семейства в круглых скобках следует индекс; тип этого индекса должен быть тем же, что и тип дискретного диапазона в соответствующем описании семейства входов. Вне тела задачи имя входа записывается в форме именованного компонента, префикс которого обозначает задачный объект, а постфикс является простым именем одного из одиночных входов или семейства входов.

Одиночный вход совмещается с подпрограммой, литералом перечисления или другим одиночным входом, если у них одинаковые идентификаторы. Совмещение для семейств входов не определено. Одиночный вход или вход семейства могут быть переименованы в процедуру, как поясняется в разд. 8.5.

Виды параметров, определенные для параметров формального раздела описания входа, такие же, как в описании подпрограммы, и имеют тот же смысл (см. 6.2). Синтаксически оператор вызова входа подобен оператору вызова процедуры; правила сопоставления параметров остаются теми же, что и для вызовов подпрограмм (см. 6.4.1 и 6.4.2).

Оператор принятия задает действия, которые выполняются при вызове упомянутого в этом операторе входа (им может быть и вход семейства). Раздел формальных параметров оператора принятия должен быть согласован с разделом формальных параметров, заданным в описании одиночного входа или семейства входов, упомянутых в операторе принятия (см. разд. 6.3.1 о согласовании). Если в конце оператора принятия используется простое имя, оно должно повторять простое имя, заданное в начале этого оператора.

Оператор принятия входа данной задачи допускается только в соответствующем теле задачи, исключая тело программного модуля, вложенного в тело задачи, и другой оператор принятия этого же одиночного входа или входа того же семейства. (Из этих правил следует, что задача может выполнять операторы принятия только своих входов). Тело задачи может содержать несколько операторов принятия одного и того же входа.

При предвыполнении описания входа вначале вычисляется дискретный диапазон (если он есть), затем также, как в описании подпрограммы, предвыполняется раздел формальных параметров (если он есть).

Выполнение оператора принятия начинается с вычисления индекса входа (в случае входа семейства). Выполнение оператора вызова входа начинается с вычисления имени входа, затем следуют вычисления, требуемые для фактических параметров, как и при вызове подпрограммы (см. 6.4). Дальнейшее выполнение оператора принятия и соответствующего оператора вызова входа синхронизировано.

Если данный вход вызывается только одной задачей, то предоставляются две возможности:

• Если вызывающая задача перешла к оператору вызова входа раньше, чем имеющая этот вход задача достигла оператора принятия, то выполнение вызывающей задачи *приостанавливается.*

• Если задача достигла оператора принятия раньше любого вызова этого входа, то выполнение задачи приостанавливается до получения такого вызова.

Если вход был вызван и соответствующий оператор принятия достигнут, то его последовательность операторов (если она есть) выполняется вызванной задачей (вызывающая задача остается приостановленной). Это взаимодействие задач называется *рандеву.* После рандеву вызывающая задача и задача, содержащая вход, продолжают выполняться параллельно.

Если несколько задач вызывают один и тот же вход до того, как достигнут оператор принятия, то эти вызовы становятся в очередь; с каждым входом связывается одна очередь. Каждое выполнение оператора принятия удаляет из очереди один вызов. Вызовы обрабатываются в порядке поступления.

При попытке вызвать вход задачи, закончившей свое выполнение, в точке вызова вызывающей задачи возбуждается исключение TASKING_ERROR; это же исключение возбуждается в точке вызова, если вызванная задача заканчивает свое выполнение до принятия входа (см. также 9.10 для случая, когда вызванная задача становится аварийной). Исключение CONSTRAINT_ERROR возбуждается, если индекс входа семейства не принадлежит заданному дискретному диапазону.

Примеры описаний входов:

```
entry ЧИТАТЬ (В: out ЭЛЕМЕНТ);
entry ЗАХВАТИТЬ;
entry ЗАПРОС (УРОВЕНЬ) (Д: ЭЛЕМЕНТ); - - семейство входов
```

Примеры вызовов входов:

```
УПРАВЛЕНИЕ. ОСВОБОДИТЬ; - - см. 9.2 и 9.1
ПОСТАВЩИК_ПОТРЕБИТЕЛЬ.ПИСАТЬ (Е); - - см. 9.1
ПУЛ(5).ЧИТАТЬ (СЛЕДУЮЩИЙ_СИМВ); - - см. 9.1 и 9.2
КОНТРОЛЛЕР. ЗАХВАТИТЬ (НИЗКИЙ) (НЕКОТОРЫЙ_ЭЛЕМЕНТ); - - см. 9.1
```

Примеры операторов принятия:

```
accept ЗАХВАТИТЬ;
accept ЧИТАТЬ (В: out ЭЛЕМЕНТ) do
      В: = ЛОКАЛЬНЫЙ_ЭЛЕМЕНТ;
end ЧИТАТЬ;
accept ЗАПРОС (НИЗКИЙ) (Д: ЭЛЕМЕНТ) do
    . . .
end ЗАПРОС;
```

Примечание. Заданный в операторе принятия раздел формальных параметров не предвыполняется; он используется только для идентификации соответствующего входа.

Оператор принятия может вызывать подпрограммы, производящие вызовы входов. Оператор принятия может не содержать последовательности операторов, даже если соответствующий вход имееет параметры. Точно также он может содержать последовательность операторов, даже если соответствующий вход не имеет параметров. Последовательность операторов в операторе принятия может включать операторы возврата. Задача может вызывать и свои собственные входы, однако это, конечно, может привести к тупиковой ситуации. Языком разрешаются условные и временные вызовы входов (см. 9.7.2 и 9.7.3). Правилами языка в данный момент времени задаче разрешается находиться только в одной очереди.

Если границы дискретного диапазона семейства входов являются литералами целого типа, то индекс (в имени входа или в операторе принятия) должен быть предопределенного типа INTEGER (см. 3.6.1).

9.6. **Операторы задержки, длительность и время**

Выполнение оператора задержки вычисляет простое выражение и приостанавливает дальнейшее выполнение задачи, содержащей оператор задержки по крайней мере на длительность, заданную вычисленным значением.

оператор_задержки : : = **delay** простое_выражение;

Простое выражение должно быть предопределенного фиксированного типа DURATION (ДЛИТЕЛЬНОСТЬ); его значение выражается в секундах; оператор задержки с отрицательным значением эквивалентен оператору задержки с нулевым значением аргумента.

Все реализации типа DURATION должны допускать представление длительности (положительные и отрицательные) по крайней мере до 86400 с (1 сут); минимальная представимая длительность, DURATION'SMALL, должна быть не больше 20 мс (по возможности, значение не превышает 50 мкс. Заметим, что DURATION'SMALL не обязано соответствовать основному циклу таймера, именованному числу SYSTEM.TICK (см. 13.7).

Определение типа TIME (ВРЕМЯ) приведено в предопределенном пакете CALENDAR (КАЛЕНДАРЬ). Функция CLOCK (ЧАСЫ) возвращает текущее значение типа TIME. Функции YEAR, MONTH, DAY и SECONDS (ГОД, МЕСЯЦ, ДЕНЬ и СЕКУНДЫ) возвращают соответствующие наименованиям значения для заданного значения аргумента типа TIME, а процедура SPLIT (СЛОЙ) возвращает одновременно все четыре соответствующих значения. Наоборот, функция TIME_OF (ВРЕМЯ_ИЗ) упаковывает номера года, месяца, числа дня и значение длительности в значение типа TIME. Операции "+" и "- " для сложения и вычитания значений времени и длительности, а также операции отношения для значений времени, имеют традиционный смысл.

Исключение TIME_ERROR (ОШИБКА_ВРЕМЕНИ) возбуждается в функции TIME_OF, если значения фактических параметров не позволяют сформировать соответствующую дату. Это исключение возбуждается также операциями "+" и "–", если для заданных операндов они не могут возвратить дату с номером года из диапазона соответствующего подтипа или если операция "–" не может возвратить значение из диапазона типа DURATION.

```
package CALENDAR is
  type TIME is private;
  subtype YEAR_NUMBER is INTEGER range 1901. .2099;
  subtype MONTH_NUMBER is INTEGER range 1. .12;
  subtype DAY_NUMBER is INTEGER range 1. .31;
  subtype DAY_DURATION is DURATION range 0.0. .86400.0;
  function CLOCK return TIME;
  function YEAR (DATE:TIME) return YEAR_NUMBER;
  function MONTH (DATE:TIME) return MONTH_NUMBER;
  function DAY (DATE:TIME) return DAY_NUMBER,
  function SECONDS (DATE:TIME) return DAY_DURATION;
  procedure SPLIT (DATE: in TIME;
                   YEAR: out YEAR_NUMBER;
                   MONTH: out MONTH_NUMBER;
                   DAY: out DAY_NUMBER;
                   SECONDS: out DAY_DURATION);
  function TIME_OF (YEAR: YEAR_NUMBER;
                    MONTH: MONTH_NUMBER;
                    DAY: DAY_NUMBER;
                    SECONDS: DAY_DURATION: = 0.0) return TIME;
  function ",+" (LEFT: TIME; RIGHT: DURATION) return TIME;
  function ",+" (LEFT: DURATION; RIGHT: TIME) return TIME;
```

```
  function "- " (LEFT:TIME; RIGHT: DURATION) return TIME;
  function "-" (LEFT:TIME; RIGHT:TIME) return DURATION;
  function "<" (LEFT,RIGHT: TIME) return BOOLEAN;
  function "<=" (LEFT, RIGHT: TIME) return BOOLEAN;
  function ">" (LEFT, RIGHT: TIME) return BOOLEAN;
  function ">=" (LEFT, RIGHT: TIME) return BOOLEAN;
  TIME_ERROR: exception; - - может быть возбуждено функцией
                         - - TIME_OF и операциями "+" и "-"
  private
  - - зависит от реализации
end;
```

Примеры:

```
delay 3.0; - - задержка на 3.0 с
declare
  use CALENDAR;
  - - ИНТЕРВАЛ – глобальная константа типа DURATION
  СЛЕДУЮЩЕЕ_ВРЕМЯ: TIME:= CLOCK + ИНТЕРВАЛ;
begin
  loop
     delay СЛЕДУЮЩЕЕ_ВРЕМЯ – CLOCK;
     - - некоторые действия
     СЛЕДУЮЩЕЕ_ВРЕМЯ: = СЛЕДУЮЩЕЕ_ВРЕМЯ + ИНТЕРВАЛ;
  end loop;
end;
```

Примечание. Во втором примере цикл повторяется в среднем один раз каждые ИНТЕРВАЛ секунд. Этот интервал между двумя последовательными итерациями только приблизителен. Однако здесь не произойдет накопление дрейфа во времени, поскольку длительность каждой итерации (существенно) меньше значения ИНТЕРВАЛ.

9.7. **Операторы отбора**

Существует три формы операторов отбора. Одна форма обеспечивает отбор с ожиданием из одной или нескольких альтернатив. Две другие обеспечивают условный и временной вызовы входа.

оператор_отбора : : = отбор_с_ожиданием
| условный_вызов_входа | временной_вызов_входа

9.7.1. Отбор с ожиданием

Эта форма оператора отбора допускает объединение ожидания и отбор одной или нескольких альтернатив. Отбор может зависеть от условий, связанных с каждой альтернативой отбора с ожиданием.

```
отбор_с_ожиданием : : =
  select
     альтернатива_отбора
  {or
     альтернатива_отбора}
  [else
     последовательность_операторов]
  end select;
альтернатива_отбора : : =
 [when условие =>]
     альтернатива_отбора_с_ожиданием
```

альтернатива_отбора_с_ожиданием : : = альтернатива_принятия
| альтернатива_задержки | альтернатива_завершения
альтернатива_принятия : : =
оператор_принятия [последовательность_операторов]
альтернатива_задержки : : =
оператор_задержки [последовательность_операторов]
альтернатива_завершения : : = **terminate**;

Оператор отбора с ожиданием должен содержать по крайней мере одну альтернативу принятия. В дополнение к этому оператор отбора с ожиданием может содержать либо альтернативу завершения (только одну), либо одну или несколько альтернатив задержки, либо раздел **else**; эти три возможности являются взаимоисключающими.

Альтернатива отбора называется *открытой*, если она не начинается с зарезервированного слова **when** или если значение условия – TRUE. В противном случае альтернатива называется *закрытой*.

При выполнении оператора отбора с ожиданием в произвольном не определенном в языке порядке вычисляются все условия, заданные после зарезервированного слова **when**; определяются открытые альтернативы. Для открытой альтернативы задержки вычисляется выражение длительности задержки. Для открытой альтернативы принятия входа семейства вычисляется индекс входа. Выполнение отбора с ожиданием заканчивается отбором и вычислением либо одной из открытых альтернатив, либо раздела **else**; правила такого отбора описываются ниже.

Первыми рассматриваются открытые альтернативы. Отбор одной из таких альтернатив производится немедленно, если возможно соответствующее рандеву, т. е. если другая задача произвела вызов соответствующего входа и ожидает его принятия. Если таким образом могут быть отобраны несколько альтернатив, то одна из них выбирается произвольно (которая именно – в языке неопределено). После отбора альтернативы выполняются соответствующий оператор принятия и следующая за ним последовательность операторов, если она есть. Если никакое рандеву немедленно не может произойти и отсутствует раздел **else**, то задача ждет, пока можно будет выбрать открытую альтернативу отбора с ожиданием.

Отбор других форм альтернатив или раздела **else** осуществляется следующим образом:

• Отбирается открытая альтернатива задержки, если никакая другая альтернатива принятия не может быть выбрана до истечения указанной задержки ожидания (немедленно, если длительность отрицательная или нулевая и при отсутствии очередей вызовов входа); затем выполняются возможные последующие операторы этой альтернативы. Если возможен отбор нескольких альтернатив задержки (т. е. если задержки у них одинаковы), то одна из них выбирается произвольно.

• Отбирается раздел **else**, выполняется последовательность операторов этого раздела, если нельзя немедленно отобрать альтернативу принятия, в частности, если все альтернативы закрыты.

• Отбирается открытая альтернатива завершения, если перечисленные в разд. 9.4 условия удовлетворены. Из других правил следует, что нельзя отобрать альтернативу завершения, пока существует очередь вызовов любого входа задачи.

Исключение PROGRAMM_ERROR возбуждается, если все альтернативы закрыты и раздел else отсутствует.

Пример оператора отбора:

```
select
  accept ВЫДАЧА_СИГНАЛА_МАШИНИСТУ;
or
  delay 30.0 * СЕКУНД;
  ОСТАНОВИТЬ_СОСТАВ;
end select;
```

Пример тела задачи с оператором отбора:

```
task body РЕСУРС is
  ЗАНЯТ: BOOLEAN: = FALSE;
begin
  loop
     select
        when not ЗАНЯТ =>
           accept ЗАХВАТИТЬ do
              ЗАНЯТ: = TRUE;
           end;
        or
           accept ОСВОБОДИТЬ do
              ЗАНЯТ: = FALSE;
           end;
        or
           terminate;
        end select;
     end loop; '
  end РЕСУРС;
```

Примечание. В отборе с ожиданием допускаются несколько открытых альтернатив задержки или несколько открытых альтернатив принятия одного и того же входа.

9.7.2. Условные вызовы входов

Условный вызов входа производит вызов входа, который отменяется, если рандеву нельзя осуществить немедленно.

```
условный_вызов_входа : : =
  select
     оператор_вызова_входа
      [последовательность_операторов]
  else
     последовательность_операторов
  end select;
```

При выполнении условного вызова входа вначале вычисляется имя входа. Затем выполняются требуемые вычисления фактических параметров, как при вызове подпрограммы (см. 6.4).

Вызов входа отменяется, если выполнение вызванной задачи не достигло точки, в которой она готова к принятию входа (т. е. не достигнуты опе-

ратор принятия соответствующего входа или оператор отбора с открытой альтернативой принятия этого входа), или существует очередь ранее сделанных вызовов этого входа. Если вызванная задача достигла оператора отбора, но альтернатива принятия этого входа не отобрана, то вызов входа отменяется.

Если вызов входа отменен, то выполняются операторы раздела else (после зарезервированного слова else). В противном случае происходит рандеву и выполняется последовательность операторов после вызова входа (если она есть).

Выполнение условного вызова входа возбуждает исключение TASKING_ERROR, если вызванная задача уже закончила свое выполнение (см. разд. 9.10 для случая, когда вызванная задача становится аварийной).

Пример:

```
procedure ОБОРОТ (Р: РЕСУРС) is
begin
  loop
     select
        Р.ЗАХВАТИТЬ;
        return;
     else;
        null; - - занято, надо подождать
     end select;
  end loop;
end;
```

9.7.3. Временные вызовы входов

Временной вызов входа производит вызов входа, который отменяется, если рандеву не началось на протяжении заданной задержки.

временной_вызов_входа : : =

```
select
   оператор_вызова_входа
    [последовательность_операторов]
or
   альтернатива_задержки
end select;
```

При выполнении временного вызова входа вначале вычисляется имя входа. Затем выполняются требуемые вычисления фактических параметров, как при вызове подпрограммы (см. разд. 6.4). После этого вычисляется выражение, задающее задержку, и, наконец, производится вызов входа.

Если рандеву может начаться в течение указанной длительности (или немедленно, как для условного вызова входа, если задержка отрицательная или нулевая), то оно происходит, и затем после вызова входа выполняется последовательность операторов (если она есть). В противном случае по истечении заданной длительности вызов входа отменяется и выполняется возможная последовательность операторов альтернативы задержки.

Выполнение временного вызова входа возбуждает исключение TASKING_ERROR, если вызванная задача закончила свое выполнение до

принятия вызова (см. также разд. 9.10 для случая, когда вызванная задача становится аварийной)

Пример:

```
select
  КОНТРОЛЛЕР.ЗАХВАТИТЬ (СРЕДНИЙ) (НЕКОТОРЫЙ_ЭЛЕМЕНТ);
or
  delay 45.0;
  - - контроллер слишком занят, попробуйте что-либо еще
end select;
```

9.8. **Приоритеты**

Каждая задача может (но не обязательно) иметь приоритет со значением подтипа PRIORITY (типа INTEGER), описанного в предопределенном библиотечном пакете SYSTEM (см. разд. 13.7). Меньшее значение приоритета указывает на меньшую степень важности; диапазон приоритетов определяется реализацией. Приоритет связывается с задачей в том случае, если в спецификации соответствующей задачи присутствует прагма:

pragma PRIORITY (статическое_выражение);

Приоритет задается значением выражения. Если такая прагма присутствует в самом внешнем разделе описаний главной программы, то приоритет связывается с главной программой. В спецификации данной задачи или для подпрограммы – библиотечного модуля может употребляться не более одной такой прагмы, и это единственные места, допустимые для этой прагмы. Прагма PRIORITY игнорируется при ее появлении в подпрограмме, не являющейся главной программой.

Спецификация приоритета является указанием, помогающим реализации в распределении ресурсов между параллельными задачами, когда число выполняемых задач превышает возможности их одновременной обработки имеющимися ресурсами. Влияние приоритетов на порядок очередности выполнения задач определяется следующим правилом:

если две задачи с разными приоритетами готовы к выполнению и могут практически выполняться, используя одни и те же физические процессоры и одни и те же ресурсы обработки, то нельзя, чтобы выполнялась задача с более низким приоритетом, а не выполнялась задача с более высоким приоритетом.

Для задач с одинаковыми приоритетами порядок выполнения в языке не определен. Для задач, приоритеты которых не заданы, правила очередности не определены, исключая случай, когда между задачами происходит рандеву. Если приоритеты обеих задач определены, то рандеву выполняется с той задачей, чей приоритет является наибольшим. Если приоритет определен только для одной из двух задач, то рандеву выполняется как минимум с приоритетной задачей. Если приоритеты задач не заданы, то приоритет рандеву также не определен.

Примечание. Приоритет задачи является статическим, и поэтому фиксирован. Однако приоритет во время рандеву может и не быть статическим, поскольку он также зависит от приоритета задачи, вызывающей вход. Приоритеты следует использовать только для указания относительной степени важности; их не следует использовать для синхронизации задач.

9.9. **Атрибуты задач и входов**

Для задачного объекта или значения T определены следующие атрибуты:
T'CALLABLE Вырабатывает значение FALSE, если выполнение указанной T задачи либо закончено, либо завершено, либо задача аварийная. В остальных случаях вырабатывает значение TRUE. Значение этого атрибута имеет предопределенный тип BOOLEAN.
T'TERMINATED Вырабатывает значение TRUE, если указанная T задача завершена. В остальных случаях вырабатывает значение FALSE. Значение этого атрибута имеет предопределенный тип BOOLEAN.

В дополнение к приведенным для задачного объекта T или задачного типа T определены атрибуты представления STORAGE_SIZE, SIZE и ADDRESS (см. 13.7.2).

Атрибут COUNT определен для входа E задачного модуля T. Вход может быть либо одиночным входом, либо входом семейства (в любом случае имя одиночного входа или семейства входов может быть либо простым, либо расширенным). Этот атрибут допустим только в теле T, но не во вложенном в тело T программном модуле.

E'COUNT Вырабатывает число вызовов входа, присутствующие в очереди входа E в данный момент (если атрибут вычисляется при выполнении оператора принятия входа E, то в это число не включается вызывающая задача). Значение атрибута имеет тип *универсальный_целый*.

Примечание. Алгоритмы, соответствующие программы которых используют атрибут E'COUNT, обязаны учитывать возможность увеличения значения атрибута с появлением новых вызовов и уменьшения этого значения, например, при временных вызовах входа.

9.10. **Операторы прекращения**

Оператор прекращения переводит одну из нескольких задач в *аварийное* состояние, предотвращая любые дальнейшие рандеву с такими задачами.

оператор_прекращения : : = **abort** имя_*задачи* {, имя_*задачи*};

При определении типа имени каждой задачи используется тот факт, что это задачный тип.

При выполнении оператора прекращения заданные имена задач вычисляются в порядке, который в языке не определен. Затем каждая упомянутая задача становится аварийной, если она еще не завершена; аналогично, любая зависящая от упомянутой задача становится также аварийной, если она еще не завершена.

Любая аварийная задача, выполнение которой приостановлено операторами принятия, отбора или задержки, становится законченной. Любая аварийная задача, выполнение которой приостановлено при вызове входа, а соответствующее рандеву еще не началось, становится законченной и удаляется из очереди ко входу. Любая аварийная задача, которая не начала свою активизацию, становится законченной (и, следовательно, также и завершенной). Этим заканчивается выполнение оператора прекращения.

Окончание любой другой аварийной задачи не производится до окончания выполнения оператора прекращения. Оно должно произойти не позже

достижения аварийной задачей точки синхронизации, которой может быть конец ее активизации, начало активизации другой задачи, вызов входа, начало или конец выполнения оператора принятия, оператор отбора, оператор задержки, обработчик исключения или оператор прекращения. Если задача, вызвавшая вход, становится аварийной в ходе рандеву, то ее завершение не производится до окончания рандеву (см. разд. 11.5).

Вызов входа аварийной задачи возбуждает в месте вызова исключение TASKING_ERROR. Аналогично, исключение TASKING_ERROR возбуждается в любой задаче, вызвавшей вход аварийной задачи, если вызов входа все еще находится в очереди, либо рандеву не окончено (вызовом входа может быть либо оператор вызова входа, либо операторы условного или временного вызова входа); исключение возбуждается не позже окончания аварийной задачи. Для любой аварийной (или законченной) задачи значение атрибута CALLABLE есть FALSE.

Если аварийное окончание задачи произошло во время изменения в задаче некоторой переменной, то значение этой переменной неопределено.

Пример:

abort ПОЛЬЗОВАТЕЛЬ, ТЕРМИНАЛ. **all**, ПУЛ (3);

Примечание. Оператор прекращения следует использовать только в крайних случаях, требующих безусловного завершения. Допускается, что задача может прекратить любую задачу, включая себя.

9.11. Разделяемые переменные

Обычными средствами передачи данных между задачами являются операторы вызова и принятия входов.

Если две задачи считывают или изменяют *разделяемую* переменную (т. е. доступную обеим задачам переменную), то ни одна из них ничего не может знать о порядке выполнения этих операций над переменной, исключая точки синхронизации. Две задачи синхронизируются в начале и в конце их рандеву. В начале и в конце своей активизации задача синхронизируется с вызвавшей эту активизацию задачей. Задача, которая закончила свое выполнение, синхронизована с любой другой задачей.

О действиях, выполняемых программой, использующей разделяемые переменные, всегда могут быть сделаны следующие предположения:

- Если в интервале времени между двумя точками синхронизации задача считывает разделяемую переменную скалярного или ссылочного типа, то эта переменная не изменяется никакой другой задачей в течение данного интервала времени.
- Если в интервале времени между двумя точками синхронизации задача изменяет разделяемую переменную скалярного или ссылочного типа, то эта переменная не считывается и не изменяется никакой другой задачей в течение данного интервала времени.

Выполнение программы ошибочно, если какое-либо из этих предположений нарушено.

Если данная задача считывает значение разделяемой переменной, сделанные выше предположения допускают, чтобы реализация поддерживала

локальные копии значения (например, в регистрах или в некоторых других видах временной памяти); и, пока данная задача не достигла точки синхронизации или не изменила значения разделяемой переменной, следствием принятых допущений является то, что для данной задачи чтение локальной копии эквивалентно чтению собственно разделяемой переменной.

Аналогично, если данная задача изменяет значение разделяемой переменной, сделанные предположения допускают, чтобы реализация поддерживала локальные копии значения и откладывала запоминание локальной копии в разделяемую переменную до точки синхронизации, заменяя каждые последующие считывание или изменение значения разделяемой переменной на считывание или изменение локальной копии. С другой стороны, не допускается, чтобы реализация вводила такую память, которая не будет обрабатываться в каноническом порядке (см. разд. 11.6).

Для задания того, что каждое считывание или изменение значения разделяемой переменной является для этой переменной точкой синхронизации, может быть использована прагма SHARED, т. е. для данной переменной (но не обязательно для остальных) сделанные выше предположения справедливы. Форма этой прагмы следующая:

pragma SHARED (простое_имя_*переменной*);

Прагма допустима только для переменной, объявленной описанием объекта скалярного или ссылочного типа; описание переменной и прагма должны помещаться (в таком порядке) непосредственно в одном и том же разделе описаний или спецификации пакета; прагма должна появиться до любого вхождения имени переменной, отличного от вхождения в спецификаторе адреса.

Реализация должна ограничивать объекты, для которых допустима прагма SHARED, объектами, для которых каждое прямое считывание или прямое изменение реализуется неделимыми операциями типа.

9.12. **Пример использования задачи**

В следующем примере определена задача буферизации для сглаживания различий между скоростью ввода производящей задачи и скоростью ввода некоторой потребляющей задачей. Например, производящая задача может содержать операторы:

```
loop
  -- выработка следующего символа СИМВ
  БУФЕР. ПИСАТЬ (СИМВ);
  exit when СИМВ = ASCII.EOT;
end loop;
```

потребляющая задача - операторы:

```
loop
  БУФЕР. ЧИТАТЬ (СИМВ);
  -- использование символа СИМВ
  exit when СИМВ = ASCII.EOT;
end loop;
```

Задача буферизации содержит внутренний пул для символов, обрабатываемых циклически. Пул имеет два индекса: ВХ_ИНДЕКС, указывающий место следующего вводимого символа, и ВЫХ_ИНДЕКС – место следующего выводимого символа.

```
task БУФЕР is
  entry ЧИТАТЬ (С: out CHARACTER);
  entry ПИСАТЬ (С: in CHARACTER);
end
task body БУФЕР is
  РАЗМЕР_ПУЛА: constant INTEGER: = 100;
  ПУЛ: array (1. .РАЗМЕР_ПУЛА) of CHARACTER;
  СЧЕТЧИК: INTEGER range 0. .РАЗМЕР_ПУЛА: = 0;
  ВХ_ИНДЕКС, ВЫХ_ИНДЕКС: INTEGER range 0. .РАЗМЕР_ПУЛА: = 1;
begin
  loop
     select
        when СЧЕТЧИК < РАЗМЕР_ПУЛА =>
           accept ПИСАТЬ (С: in CHARACTER) do
              ПУЛ (ВХ_ИНДЕКС): = С;
           end;
           ВХ_ИНДЕКС: = ВХ_ИНДЕКС mod РАЗМЕР_ПУЛА + 1;
           СЧЕТЧИК: = СЧЕТЧИК + 1;
     or when СЧЕТЧИК > 0 =>
           accept ЧИТАТЬ (С: out CHARACTER) do
              С: = ПУЛ (ВЫХ_ИНДЕКС);
           end;
           ВЫХ_ИНДЕКС: = ВЫХ_ИНДЕКС mod РАЗМЕР_ПУЛА + 1;
           СЧЕТЧИК: = СЧЕТЧИК –1;
     or
        terminate;
     end select;
  end loop;
end БУФЕР;
```

10. СТРУКТУРА ПРОГРАММЫ И РЕЗУЛЬТАТ КОМПИЛЯЦИИ

В этой главе описываются общая структура программы и средства раздельной компиляции. Программа представляет собой набор из одного или нескольких компилируемых модулей, подаваемых на вход компилятора в виде одной или нескольких компиляций. Каждый компилируемый модуль определяет раздельную компиляцию некоторой конструкции. Ею может быть описание или тело подпрограммы, описание или тело пакета, описание или тело настраиваемого модуля или же конкретизация настройки. Кроме того, компилируемый модуль может быть субмодулем, т. е. телом подпрограммы, пакета, задачи или настраиваемого модуля, описанных внутри другого компилируемого модуля.

10.1. **Компилируемые модули. Библиотечные модули**

Текст программы может подаваться на вход компилятора в виде одной или нескольких компиляций. Каждая компиляция представляет собой последовательность компилируемых модулей.

```
компиляция : : = {компилируемый_модуль}
компилируемый_модуль : : =
  спецификатор_контекста  библиотечный_модуль
  | спецификатор_контекста  вторичный_модуль
библиотечный_модуль : : = описание_подпрограммы
```

| описание_пакета | описание_настройки
| конкретизация_настройки | тело_подпрограммы
вторичный_модуль : : =
тело_библиотечного_модуля | субмодуль
тело_библиотечного_модуля : : =
тело_подпрограммы | тело_пакета

Говорят, что компилируемые модули программы принадлежат *программной библиотеке.* Компилируемый модуль определяет библиотечный модуль или вторичный модуль. Вторичный модуль – это раздельно компилируемое соответствующее тело библиотечного модуля или субмодуль другого компилируемого модуля. Обозначением раздельно компилируемой подпрограммы (библиотечного модуля или субмодуля) должен быть идентификатор. В программной библиотеке простые имена всех библиотечных модулей должны быть различными идентификаторами.

Результат компилирования библиотечного модуля состоит в том, чтобы определить (или переопределить) его как модуль программной библиотеки. По правилам видимости каждый библиотечный модуль рассматривается как описание, приведенное непосредственно внутри пакета STANDARD.

Результат компилирования вторичного модуля состоит в том, чтобы определить тело библиотечного модуля или, в случае субмодуля, определить соответствующее тело программного модуля, описанного внутри другого компилируемого модуля.

Тело подпрограммы в качестве компилируемого модуля рассматривается как вторичный модуль в том случае, если программная библиотека уже содержит библиотечный модуль, который является подпрограммой с тем же именем. В противном случае оно рассматривается одновременно и как библиотечный модуль, и как соответствующее этому библиотечному модулю тело (т. е. как вторичный модуль).

Компилируемые модули одной компиляции транслируются в заданном порядке. Относящаяся ко всей компиляции прагма должна помещаться перед первым компилируемым модулем компиляции.

Подпрограмма, являющаяся библиотечным модулем, может использоваться в качестве *главной программы* в традиционном смысле. Главная программа выполняется так, как будто она вызвана некоторой внешней задачей; средства инициализации этого выполнения в языке не предписаны. Реализация может предъявить определенные требования к параметрам и результату (если он есть) главной программы (эти требования должны быть приведены в руководствах в соответствии с обязательным приложением 4). Каждая реализация должна разрешать задание в качестве главной программы процедуры без параметров. Любая главная программа должна быть подпрограммой – библиотечным модулем.

Примечание. Простая программа может состоять из одного компилируемого модуля. Компиляция может не содержать ни одного компилируемого модуля, например, ее текст может состоять из одних прагм.

Обозначение библиотечной функции не может быть знаком операции, но для переименования библиотечной функции в операцию допускается использовать описание

переименования. Две библиотечные подпрограммы должны иметь различные простые имена, и потому не могут быть совмещены друг с другом. Однако описания переименования могут определить совмещенные имена для таких подпрограмм; кроме того, с библиотечной подпрограммой можно совмещать локально описанную подпрограмму. Так как библиотечные модули рассматриваются как описания, приведенные непосредственно в пакете STANDARD, то для библиотечного модуля M может быть использовано расширенное имя STANDARD.M (если только имя STANDARD не скрыто).

10.1.1. Спецификаторы контекста. Спецификаторы совместности

Для указания библиотечных модулей, имена которых используются в компилируемом модуле, служит спецификатор контекста.

спецификатор_контекста : : =

{спецификатор_совместности {спецификатор_ использования}}

спецификатор_совместности : : =

wiht простое_имя_*модуля* { простое_имя_*модуля*};

Имена в спецификаторе контекста должны быть простыми именами библиотечных модулей. В спецификаторе совместности допустимы простые имена любых библиотечных модулей. В спецификаторе использования допустимы простые имена только тех библиотечных пакетов, которые указаны в предшествующих спецификаторах совместности данного спецификатора контекста. В спецификаторе контекста недопустимы простые имена, введенные описанием переименования.

Спецификаторы совместности и использования в спецификаторе контекста библиотечного модуля *применяются* к этому библиотечному модулю и ко вторичному модулю, который определяет соответствующее тело (независимо от того, повторен ли такой спецификатор для вторичного модуля или нет). Аналогично, для компилируемого модуля спецификаторы совместности и использования в спецификаторе контекста *применяются* к этому модулю и к его субмодулям (если они есть).

Библиотечный модуль, упомянутый в спецификаторе совместности, примененном к компилируемому модулю, видим непосредственно внутри этого компилируемого модуля, исключая случаи его скрытия; этот библиотечный модуль видим как описанный непосредственно в пакете STANDARD (см. 8.6).

Спецификаторы совместности задают зависимости между компилируемыми модулями, т. е. компилируемый модуль зависит от других библиотечных модулей, упомянутых в спецификаторе контекста. Зависимости между модулями учитываются при определении допустимого порядка компиляции (и перекомпиляции) компилируемых модулей (см. разд. 10.3), а также при определении допустимого порядка предвыполнения компилируемых модулей (см. разд. 10.5).

Примечание. Внутри компилируемого модуля видимы библиотечные модули, упомянутые в спецификаторе совместности (исключая случаи их скрытия), следовательно, они могут использоваться как соответствующие программные модули. Так, в компилируемом модуле имя библиотечного пакета может быть дано в спецификаторе использования и служит для формирования расширенных имен; библиотечные под-

программы можно вызывать; можно описать конкретизации библиотечного настраиваемого модуля.

Правила для спецификаторов совместности дают одинаковый результат как при упоминании имени библиотечного модуля один или несколько раз в различных спецификаторах совместности, так и внутри заданного спецификатора совместности.

Пример 1. Главная программа. Ниже приведен пример главной программы, состоящей из одного компилируемого модуля – процедуры печати вещественных корней квадратного уравнения. Предполагается, что в программной библиотеке уже содержится предопределенный пакет TEXT_IO и заданный пользователем пакет ВЕЩЕСТВ_ОПЕРАЦИИ (содержащий определения типа ВЕЩЕСТВ и пакетов ВЕЩЕСТВ_ВВ и ВЕЩЕСТВ_ ФУНКЦИИ).

Такие пакеты могут быть использованы и другими главными программами.

```
with TEXT_IO,ВЕЩЕСТВ_ОПЕРАЦИИ;
use ВЕЩЕСТВ_ОПЕРАЦИИ;
procedure КВАДРАТНОЕ_УРАВНЕНИЕ is
  A, B, C, Д: ВЕЩЕСТВ;
  use ВЕЩЕСТВ_ВВ - - обеспечивает прямую видимость GET и PUT для ВЕЩЕСТВ
  TEXT_IO, - - обеспечивает прямую видимость NEW_LINE и PUT для строк
  ВЕЩЕСТВ_ФУНКЦИИ; - - обеспечивает прямую видимость функции SQRT
begin
  GET (A); GET (B); GET (C);
  Д: = B ** 2 – 4.0 * A * C;
  if Д < 0.0 then
  PUT ("МНИМЫЕ КОРНИ");
  else
     PUT ("ВЕЩЕСТВЕННЫЕ КОРНИ: X1 =");
     PUT((–B –SQRT (Д)) / (2.0 * A)); PUT("X2 =");
     PUT ((–B +SQRT(Д)) / (2.0 * A));
  end if;
   NEW_LINE,
end КВАДРАТНОЕ_УРАВНЕНИЕ;
```

Примечание к примеру. В спецификаторах совместности компилируемого модуля надо упоминать имена только тех библиотечных подпрограмм или пакетов, видимость которых действительно необходима внутри модуля. Нет необходимости (и не следует) упоминать имена других библиотечных модулей, используемых внутри модулей, перечисленных в этих спецификаторах совместности, кроме тех, которые используются непосредственно в данном компилируемом модуле. Например, в теле пакета ВЕЩЕСТВ_ОПЕРАЦИИ могут потребоваться некоторые элементарные операции, определенные в других пакетах. Но эти пакеты не надо упоминать в спецификаторе совместности процедуры КВАДРАТНОЕ_УРАВНЕНИЕ, так как в ее теле элементарные операции не вызываются непосредственно.

10.1.2. Примеры компилируемых модулей

Компилируемый модуль может быть расчленен на несколько компилируемых модулей. Например, рассмотрим следующую программу.

```
procedure ПРОЦЕССОР is
  МАЛОЕ: constant: = 20;
  ИТОГ: INTEGER. = 0;
  package ФОНД is
     ПРЕДЕЛ: constant: = 1000;
```

```
      ТАБЛИЦА: array (1. .ПРЕДЕЛ) of INTEGER;
      procedure ПЕРЕЗАПУСК;
   end ФОНД;
   package body ФОНД is
      procedure ПЕРЕЗАПУСК is
      begin
         for К in 1. .ПРЕДЕЛ loop
            ТАБЛИЦА (К): = К;
         end loop,
      end;
begin
      ПЕРЕЗАПУСК;
end ФОНД;
procedure ИЗМЕНЕНИЕ (Х: INTEGER) is
      use ФОНД;
      - - некоторые описания
begin
      . . .
      ТАБЛИЦА (Х): = ТАБЛИЦА (Х) + МАЛОЕ;
      . . .
      end ИЗМЕНЕНИЕ;
   begin
   . . .
   ФОНД.ПЕРЕЗАПУСК; - - переинициализация массива ТАБЛИЦА
   . . .
end ПРОЦЕССОР;
```

Приведенные ниже три компилируемых модуля определяют программу с результатом, эквивалентным результату предыдущего примера (пунктирные линии между модулями напоминают, что они не обязаны составлять единый текст).

Пример 2. Несколько компилируемых модулей:

```
package ФОНД is
   ПРЕДЕЛ: constant: = 1000;
   ТАБЛИЦА: array (1. .ПРЕДЕЛ) of INTEGER;
   procedure ПЕРЕЗАПУСК;
end ФОНД;
```

```
package body ФОНД is
   procedure ПЕРЕЗАПУСК is
   begin
      for К in 1. .ПРЕДЕЛ loop
         ТАБЛИЦА (К): = К;
      end loop;
   end;
begin
   ПЕРЕЗАПУСК;
end ФОНД;
```

```
with ФОНД;
procedure ПРОЦЕССОР is
   МАЛОЕ: constant: = 20;
   ИТОГ: INTEGER: = 0;
   procedure ИЗМЕНЕНИЕ (Х: INTEGER) is
      use ФОНД;
```

```
    begin
      ...
      ТАБЛИЦА(X) := ТАБЛИЦА(X) + МАЛОЕ;
      ...
    end ИЗМЕНЕНИЕ;
  begin
    ...
    ФОНД.ПЕРЕЗАПУСК; -- переинициализация ТАБЛИЦА
    ...
  end ПРОЦЕССОР;
```

Заметим, что в последней версии примера в пакете ФОНД не видимы внешние идентификаторы, отличные от предопределенных (в пакете STANDARD). В частности, в нем не используются идентификаторы МАЛОЕ и ИТОГ, описанные в процедуре ПРОЦЕССОР; в противном случае пакет ФОНД нельзя выделить из процедуры ПРОЦЕССОР, как это сделано выше. С другой стороны, процедура ПРОЦЕССОР зависит от пакета ФОНД и упоминает его в спецификаторе совместности. Поэтому пакет ФОНД можно использовать в расширенном имени и в спецификаторе использования.

Эти три компилируемых модуля могут быть организованы как одна или несколько компиляций. Например, возможно объединение в одной компиляции спецификации и тела пакета в указанном порядке.

10.2. **Субмодули компилируемых модулей**

Субмодули используются для раздельной компиляции соответствующего тела программного модуля, описанного в другом компилируемом модуле. Этот метод разделения программы позволяет разрабатывать программу иерархически.

след_тела : : =
  спецификация_подпрограммы **is separate**;
 |**package body** простое_имя_*пакета* **is separate**;
 |**task body** простое_имя_*задачи* **is separate**;
субмодуль : : =
  separate (имя_*родительского_модуля*) соответствующее_тело

Использование следа тела в качестве тела программного модуля (подпрограммы, пакета, задачного модуля или настраиваемого модуля) допускается, только если след тела помещен непосредственно в теле библиотечного пакета или в разделе описаний некоторого компилируемого модуля.

В случае задания тела программного модуля следом тела требуется, чтобы субмодуль, содержащий соответствующее тело, был откомпилирован раздельно. Для подпрограммы спецификации подпрограммы, данные в соответствующем теле и в следе тела, должны быть согласованы (см. 6.3.1).

Для каждого субмодуля задается имя *родительского модуля*, т. е. компилируемого модуля, содержащего соответствующий след тела. Если родительский модуль – библиотечный модуль, то он называется *предком*. Если родительский модуль сам является субмодулем, то его имя должно быть представлено расширенным именем, начинающимся простым именем библиотечного модуля-предка. Простые имена всех субмодулей, которые имеют одинакового предка, должны задаваться различными идентификаторами.

Видимость в соответствующем теле субмодуля – это видимость, которая была бы получена в месте задания следа тела (в родительском модуле), если бы спецификаторы совместности и использования субмодуля были добавлены к спецификатору контекста родительского модуля. Если родительский модуль сам является субмодулем, то это же правило используется для определения видимости в соответствующем теле родительского модуля.

Результатом предвыполнения следа тела является предвыполнение соответствующего тела субмодуля.

Примечание. Два субмодуля различных библиотечных модулей в одной и той же программной библиотеке могут иметь совпадающие идентификаторы. В этом случае их расширенные имена различны, так как различны простые имена библиотечных модулей и простые имена всех субмодулей одного предка-данного библиотечного модуля. Средствами описаний переименования могут быть введены для (различных) субмодулей совмещенные имена подпрограмм.

Библиотечный модуль, упомянутый в спецификаторе совместности субмодуля, может быть скрыт описанием (с тем же идентификатором), данным в соответствующем теле субмодуля. Более того, такой библиотечный модуль может быть скрыт описанием, данным в родительском модуле, так как библиотечный модуль рассматривается как описанный в пакете STANDARD; это, однако, не влияет на интерпретацию спецификаторов совместности, ибо в них могут быть упомянуты имена только библиотечных модулей.

10.2.1. Примеры субмодулей

В первом примере процедура ВЕРШИНА оформлена в виде компилируемого модуля без субмодулей.

```
with TEXT_IO;
procedure ВЕРШИНА is
type ВЕЩЕСТВ is digits 10;
P, C: ВЕЩЕСТВ: = 1.0;
package СЕРВИС is
  ПИ: constant: = 3.14159_26536;
  function F (X: ВЕЩЕСТВ) return ВЕЩЕСТВ;
  procedure G(Y, H: ВЕЩЕСТВ) ;
end СЕРВИС;
package body СЕРВИС is
  - - предшествуют некоторые локальные описания
  function F (X: ВЕЩЕСТВ) return ВЕЩЕСТВ is
  begin
  - - последовательность операторов функции F
  . . .
  end F;
  procedure G(Y, H: ВЕЩЕСТВ) is
  - - использующие пакет TEXT_IO локальные процедуры
  . . .
  begin
  - - последовательность операторов процедуры G
  . . .
  end G;
end СЕРВИС;
procedure ПРЕОБРАЗОВАНИЕ (Ф: in out ВЕЩЕСТВ) is
use СЕРВИС;
```

```
begin
Ф: = F (Ф);
...
end ПРЕОБРАЗОВАНИЕ;
begin - - ВЕРШИНА
  ПРЕОБРАЗОВАНИЕ (Р);
  ...
  СЕРВИС.G (P, C);
end ВЕРШИНА;
```

Тело пакета СЕРВИС и процедуру ПРЕОБРАЗОВАНИЕ можно представить в виде раздельно компилируемых субмодулей модуля ВЕРШИНА. Тело процедуры G также может быть представлено как субмодуль модуля СЕРВИС.

Пример 3:

```
procedure ВЕРШИНА is
  type ВЕЩЕСТВ is digits 10;
  P, C: ВЕЩЕСТВ: = 1.0;
  package СЕРВИС is
     ПИ: constant: = 3.14159_26536;
     function F (X: ВЕЩЕСТВ) return ВЕЩЕСТВ;
     procedure G(Y, H: ВЕЩЕСТВ);
  end СЕРВИС;
  package body СЕРВИС is separate; - - след тела СЕРВИС
  procedure ПРЕОБРАЗОВАНИЕ (Ф: in out ВЕЩЕСТВ) is separate
begin - - ВЕРШИНА
  ПРЕОБРАЗОВАНИЕ (Р);
  ...
  СЕРВИС.G (P, C);
end ВЕРШИНА;
```

```
separate (ВЕРШИНА)
procedure ПРЕОБРАЗОВАНИЕ (Ф: in out ВЕЩЕСТВ) is
  use СЕРВИС;
begin
  Ф: = F (Ф);
  ...
end ПРЕОБРАЗОВАНИЕ;
```

```
separate (ВЕРШИНА)
package body СЕРВИС is
  - - предшествуют некоторые локальные описания
  function F (X: ВЕЩЕСТВ) return ВЕЩЕСТВ is
  begin
     - - последовательность операторов функции F
     ...
  end;
  procedure G(Y, H: ВЕЩЕСТВ) is separate; - - след тела G
end СЕРВИС;
```

```
with TEXT_IO;
separate (ВЕРШИНА, СЕРВИС) - - полное имя пакета СЕРВИС
procedure G(Y, H: ВЕЩЕСТВ) is
  - - использующие TEXT_IO локальные процедуры
  ...
```

```
begin
  - - последовательность операторов процедуры G
  . . .
end G;
```

В этом примере ПРЕОБРАЗОВАНИЕ и СЕРВИС являются субмодулями процедуры ВЕРШИНА, а G — субмодулем пакета СЕРВИС. Видимость в этом примере такая же, как и в предыдущем, с одним отличием: TEXT_IO используется только в G, поэтому соответствующий спецификатор совместности написан для G, а не для процедуры ВЕРШИНА. В остальном видимость идентификаторов в соответствующих телах программ обеих версий одинакова. Например, в соответствующем теле субмодуля G (непосредственно) видимы процедура ВЕРШИНА, тип ВЕЩЕСТВ, переменные P и C, пакет СЕРВИС и содержащиеся в нем именованное число ПИ и подпрограммы F и G.

10.3. **Порядок компиляции**

Правила, определяющие порядок компилирования модулей, являются непосредственным следствием правил видимости и, в частности, того факта, что любой библиотечный модуль, упомянутый в спецификаторе контекста компилируемого модуля, видим в нем.

Компилируемый модуль должен компилироваться после всех библиотечных модулей, указанных в его спецификаторе контекста. Вторичный модуль, являющийся телом подпрограммы или пакета, должен компилироваться после соответствующего библиотечного модуля. Любой субмодуль родительского компилируемого модуля должен компилироваться после него.

Компилирование, при котором в компилируемом модуле обнаружена ошибка, считается неудавшимся и никак не влияет на программную библиотеку; то же самое относится и к перекомпиляции (никакой компилируемый модуль не может стать устаревшим вследствие такой перекомпиляции).

Порядок компилирования компилируемых модулей программы должен отвечать частичной упорядоченности, определенной приведенными выше правилами.

Аналогичные правила применяются при перекомпиляции. На компилируемый модуль потенциально влияет изменение любого библиотечного модуля, упомянутого в его спецификаторе контекста. На вторичный модуль потенциально влияет изменение соответствующего библиотечного модуля. На субмодуль потенциально влияет изменение родительского компилируемого модуля. В результате успешной перекомпиляции компилируемого модуля все компилируемые модули, на которые потенциально влияет данный, считаются устаревшими и должны перекомпилироваться, кроме тех, которые больше не нужны. Реализация может уменьшить стоимость перекомпиляции, если установит, что данные изменения не повлияли на модули, которые находились под потенциальным влиянием данного.

Перекомпилирование субмодулей некоторого модуля на сам этот модуль никакого влияния не оказывает. Изменения в теле подпрограммы или

пакета не влияют на другие компилируемые модули (кроме субмодулей этого тела), так как эти компилируемые модули имеют доступ только к спецификации подпрограммы или пакета. В реализации допустимы описанные ниже отступления от этого правила, связанные с открытыми подстановками, некоторыми оптимизациями, осуществляемыми компилятором, и некоторыми аспектами реализации настраиваемых программных модулей.

• Если прагма INLINE применяется к описанию подпрограммы в спецификации пакета, то открытая подстановка возможна, только если тело пакета откомпилировано раньше модулей, вызывающих эту подпрограмму. В таком случае открытая подстановка создает *зависимость* вызывающего модуля от тела пакета; компилятор должен распознавать такую зависимость при определении необходимости перекомпиляции. Если вызывающий модуль компилируется раньше, чем тело пакета, то для таких вызовов прагма INLINE может игнорироваться компилятором (при этом может быть выдано предупреждение о невозможности открытой подстановки). То же относится и к раздельно компилируемой подпрограмме, к которой применена прагма INLINE.

• С целью оптимизации реализация может компилировать несколько модулей данной компиляции, создавая при этом дальнейшие зависимости между этими компилируемыми модулями. Эти зависимости должны учитываться компилятором для определения необходимых перекомпиляций.

• Реализация может требовать, чтобы описание настройки и соответствующее тело были частями одной и той же компиляции независимо от того, раздельно ли компилируется настраиваемый модуль, или он локален в другом компилируемом модуле. Реализация может также требовать, чтобы субмодули настраиваемого модуля были частью одной и той же компиляции.

Примеры порядка компиляции:

а. В примере 1 (см. 10.1.1) процедура КВАДРАТНОЕ_УРАВНЕНИЕ должна компилироваться после библиотечных пакетов TEXT_IO и ВЕЩЕСТВ_ОПЕРАЦИИ, так как они упомянуты в спецификаторе совместности процедуры.

б. В примере 2 (см. 10.1.2) тело пакета ФОНД должно компилироваться после соответствующей спецификации пакета.

в. В примере 2 (см. 10.1.2) спецификация пакета ФОНД должна компилироваться до процедуры ПРОЦЕССОР. С другой стороны, процедура ПРОЦЕССОР может компилироваться как до, так и после тела пакета ФОНД.

г. В примере 3 (см. 10.2.1) процедура G должна компилироваться после пакета TEXT_IO, так как этот пакет упомянут в спецификаторе совместности процедуры G. В то же время пакет TEXT_IO может компилироваться как до, так и после процедуры ВЕРШИНА.

д. В примере 3 (см. 10.2.1) субмодули ПРЕОБРАЗОВАНИЕ и СЕРВИС должны компилироваться после главной программы ВЕРШИНА. Субмодуль G должен компилироваться после его родительского модуля СЕРВИС.

Примечание. Для библиотечных пакетов из правил перекомпиляции следует, что тело пакета становится устаревшим после перекомпиляции соответствующей спецификации. Если новая спецификация пакета не требует задания тела (т. е. она не содержит описаний программных модулей), то перекомпиляции тела такого пакета не требуется. В любом случае устаревшее тело пакета не должно использоваться, и поэтому может быть удалено из программной библиотеки.

10.4. **Программная библиотека**

Правила языка требуют, чтобы компилятор одинаковым образом обрабатывал программу, состоящую из нескольких компилируемых модулей (и субмодулей) или из одного компилируемого модуля. Должен быть предусмотрен библиотечный файл, содержащий информацию о компилируемых модулях программной библиотеки, в который могут включаться символьные таблицы и другая информация, относящаяся к предыдущим компиляциям.

Обычно входными данными для компилятора являются компилируемые модули (или модуль) и библиотечный файл. Последний используется для проверок и корректируется после успешного компилирования этих модулей.

Примечание. Компилируемые модули компиляции попадают в одну программную библиотеку. Возможно существование различных программных библиотек; в языке не определены правила их именования – это обеспечивается окружением системы программирования.

Для создания программной библиотеки данной программы или данного семейства программ следует ввести команды. Эти команды могут разрешать использование модулей из других программных библиотек. Наконец, для запроса состояний модулей в программной библиотеке также следует ввести команды. Форма этих команд не задана в определении языка.

10.5. **Предвыполнение библиотечных модулей**

Перед выполнением главной программы все библиотечные модули, необходимые для главной программы, и тела этих модулей (если они есть) предвыполняются. Такими библиотечными модулями являются модули, упомянутые в спецификаторах совместности для главной программы, ее тела и субмодулей, а также модули, упомянутые в спецификаторах совместности для этих библиотечных модулей, их тел и субмодулей, и так далее, вплоть до получения транзитивного замыкания.

Предвыполнение таких библиотечных модулей и их тел производится в соответствии с частичной упорядоченностью, определяемой спецификаторами совместности (см. 10.3). Кроме того, библиотечный модуль, упомянутый в спецификаторе контекста для субмодуля, должен быть предвыполнен до тела библиотечного модуля-предка этого субмодуля.

Порядок предвыполнения, отвечающий такому отношению упорядоченности, не всегда обеспечивает выполнение следующего требования: тело любого библиотечного модуля предвыполняется прежде любого другого компилируемого модуля, при предвыполнении которого необходимо предвыполнение тела этого библиотечного модуля. Для указания необходимости более раннего предвыполнения тел библиотечных модулей используется прагма ELABORATE (ПРЕДВЫПОЛНЕНИЕ). Эта прагма записывается в виде:

pragma ELABORATE (простое_имя_*библиотечного_модуля*
{, простое_имя_*библиотечного_модуля*});

Такие прагмы допустимы только непосредственно после спецификатора контекста компилируемого модуля (до следующего за ним библиотечного или вторичного модуля). Аргументы этой прагмы должны быть простыми именами библиотечных модулей, упомянутых в спецификаторе контекста, и каждый такой библиотечный модуль должен иметь соответствующее тело. Эта прагма указывает, что тело библиотечного модуля должно предвыполняться до данного компилируемого модуля. Если данный компилируемый модуль – субмодуль, то тело библиотечного модуля должно предвыполняться до тела библиотечного модуля-предка (субмодуля).

Программа неправильна, если не может быть найден необходимый порядок предвыполнения (т. е. если существуют циклические зависимости). Предвыполнение компилируемых модулей программы в остальных случаях осуществляется в некотором порядке, который не определен в языке.

10.6. **Оптимизация программы**

Компиляторы могут осуществлять оптимизацию предвыполнения описаний и выполнения операторов. В частности, компилятор может оптимизировать программу, вычисляя определенные выражения помимо статических. Если одно из таких выражений, статических или нет, при вычислении привело бы к возбуждению исключения, то код этой части программы может быть заменен кодом возбуждения того же исключения; это справедливо для исключений, возбуждаемых при вычислении имен и простых выражений (см. также разд. 11.6).

Компилятор может определить, что некоторые операторы или подпрограммы никогда не будут выполняться, например, если их выполнение зависит от условия, имеющего значение FALSE. В таком случае соответствующие части объективного машинного кода могут быть опущены. Такое правило позволяет на уровне языка производить *условную компиляцию.*

Примечание. Выражение, вычисление которого может привести к возбуждению исключения, не обязательно представляет ошибку, если выражение находится в операторе или подпрограмме, которые никогда не выполняются. Компилятор может предупреждать программиста о потенциальной ошибке.

11. ИСКЛЮЧЕНИЯ

В этой главе определяются средства обработки ошибок или других исключительных ситуаций, которые возникают при выполнении программы. Такая ситуация называется *исключением. Возбуждение* исключения следует понимать как прекращение нормального выполнения программы для обработки соответствующей ситуации. Ответное действие на возбуждение исключения называется *обработкой* исключения.

В описании исключения задается имя исключения. Исключение может быть возбуждено либо оператором возбуждения, либо каким-либо другим оператором или операцией, *распространяющими* исключение. При возбуж-

дении исключения управление может быть передано обработчику исключения, написанному пользователем либо в конце оператора блока, либо в конце тел подпрограммы, пакета или задачного модуля.

11.1. **Описания исключений**

В описании исключения задается его имя. Это имя можно употреблять только в операторах возбуждения, обработчиках исключений и описаниях переименования.

описание_исключения : : = *список_идентификаторов*: **exception**;

Описание исключения с несколькими идентификаторами эквивалентно последовательности единичных описаний с одним идентификатором (см. разд. 3.2). Каждое единичное описание исключения задает имя отличного от других исключения. В частности, если настраиваемый модуль содержит описание исключения, то такие описания, неявно сгенерированные различными конкретизациями настройки, ссылаются на разные исключения (но все они имеют один и тот же идентификатор). Конкретное исключение, обозначенное именем, определяется во время компиляции и является одним и тем же, независимо от числа предвыполнений его описания. Следовательно, если описание исключения находится в рекурсивной подпрограмме, то имя исключения обозначает одно и то же исключение для всех вызовов рекурсивной подпрограммы.

Следующие исключения предопределены в языке. Они возбуждаются при обнаружении описанных ниже ситуаций (в скобках дан перевод имени исключения на русский язык).

CONSTRAINT_ERROR (ОШИБКА_ОГРАНИЧЕНИЯ) Это исключение возбуждается в любой из следующих ситуаций: при попытке нарушить ограничение диапазона, ограничение индекса или ограничение дискриминанта; при попытке использовать компонент записи, не существующий для текущих значений дискриминанта; при попытке использовать именуемый компонент, индексируемый компонент, отрезок или атрибут объекта, обозначенного ссылочным значением, если этот объект не существует, поскольку ссылочное значение равно **null**.

NUMERIC_ERROR (ЧИСЛОВАЯ_ОШИБКА) Это исключение возбуждается при выполнении предопределенной числовой операции, которая не может выработать верный результат (для вещественных типов в пределах предписанной точности). Сюда относится также случай использования реализацией предопределенных числовых операций для выполнения, вычисления или предвыполнения некоторой конструкции. Данные в разд. 4.5.7 правила определяют случаи, для которых от реализации не требуется возбуждения исключений при возникновении этой ошибочной ситуации (см. также разд. 11.6).

PROGRAM_ERROR (ПРОГРАММНАЯ_ОШИБКА) Это исключение возбуждается при попытке вызвать подпрограмму, активизировать задачу или предвыполнить конкретизацию настройки, если тело соответствующего модуля еще не предвыполнено. Оно также возбуждается, если выполнение функции завершается через **end** (см. разд. 6.5); при выполнении оператора

отбора с ожиданием, когда отсутствует раздел **else**, а все альтернативы закрыты (см. разд. 9.7.1). Наконец, в зависимости от реализации исключение может возбуждаться при попытке выполнить ошибочное действие и при некорректной зависимости от порядка (см. разд. 1.6).

STORAGE_ERROR (ОШИБКА_ПАМЯТИ) Это исключение возбуждается в одной из следующих ситуаций: при вычислении генератора, когда не хватает динамической памяти, выделенной для задачи; когда исчерпана память для набора генерируемых объектов; при предвыполнении элемента описания или при вызове подпрограммы, если памяти недостаточно.

TASKING_ERROR (ОШИБКА_ЗАДАЧИ) Это исключение возбуждается при возбуждении исключений во время взаимодействия задач (см. разд. 9 и 11.5).

Примечание. Описанные выше ситуации могут возникать, не возбуждая соответствующих исключений, если была использована прагма SUPPRESS подавления проверок (см. разд. 11.7).

Примеры описанных пользователем исключений:

ВЫРОЖДЕННАЯ: **exception**;
ОШИБКА: **exception**;
ПЕРЕПОЛНЕНИЕ, ПОТЕРЯ_ТОЧНОСТИ: **exception**;

11.2. **Обработчики исключений**

Ответная реакция на одно или несколько исключений определяется обработчиком исключения.

обработчик_исключения : : =
when выбор_исключения {| выбор_исключения} =>
последовательность_операторов
выбор_исключения : : = имя_*исключения* | **others**

Обработчик исключения помещается в операторе блока, в теле подпрограммы, пакета, задачного модуля или настраиваемого модуля. Каждая из указанных конструкций называется в этой главе *окружением*. В любом случае окружение обработчиков исключений синтаксически включает следующий раздел:

begin
последовательность_операторов
exception
обработчик_исключения
{обработчик_исключения}
end

Исключения, обозначенные именами исключений во всех выборах данного окружения, должны быть все различны. Выбор исключения **others** допустим только в последнем обработчике исключения данного окружения, как единственный выбор; он задает все исключения, не перечисленные в предыдущих обработчиках окружения, включая исключения, имена которых на месте данного обработчика исключения невидимы.

Обработчики исключений в окружении обрабатывают исключения, возбуждаемые при выполнении последовательности операторов этого окруже-

ния. Исключения обрабатываются тем обработчиком, в выборах которого заданы имена этих исключений.

Пример:

```
begin
  - - последовательность операторов
exception
  when ВЫРОЖДЕННАЯ | NUMERIC_ERROR =>
     PUT ("ВЫРОЖДЕННАЯ_МАТРИЦА");
  when others =>
     PUT ("НЕИСПРАВИМАЯ_ОШИБКА");
     raise ОШИБКА;
end;
```

Примечание. В каждом обработчике исключений и в последовательностях операторов окружения допустимы одни и те же виды операторов. Например, оператор возврата допустим в обработчике, если окружение является телом функции.

11.3. **Операторы возбуждения**

Оператор возбуждения возбуждает исключение.

оператор_возбуждения : : = **raise** [имя_*исключения*] ;

При выполнении оператора возбуждения с именем исключения возбуждается заданное исключение. Оператор возбуждения без имени исключения допустим только внутри обработчика исключения (но не в последовательности операторов подпрограммы, пакета, задачного модуля или настраиваемого модуля, вложенной в данный обработчик); он снова возбуждает то же исключение, которое вызвало переход на обработчик, содержащий данный оператор возбуждения.

Примеры:

```
raise ВЫРОЖДЕННАЯ;
raise NUMERIC_ERROR; - - явно возбуждается предопределенное исключение
raise; - - только внутри обработчика исключения
```

11.4. **Обработка исключения**

После возбуждения исключения нормальное выполнение программы прекращается и управление передается обработчику исключения. Выбор обработчика зависит от места возбуждения: при выполнении операторов или при предвыполнении описаний.

11.4.1. Исключения, возбуждаемые при выполнении операторов

Обработка исключения, возбуждаемого при выполнении последовательности операторов, зависит от того, вложена ли она в самое внутреннее окружение или в оператор принятия. Случай вложенности в оператор принятия описан в разд. 11.5.

Предпринимаемые действия зависят от того, содержит ли данное окружение обработчик этого исключения и возбуждено ли исключение в последовательности операторов окружения или обработчика исключения.

Если некоторое исключение возбуждено в последовательности операторов окружения, содержащего нужный обработчик, то выполнение этой последовательности операторов прекращается и управление передается обработчику данного исключения. Выполнение последовательности операторов

обработчика заканчивает выполнение окружения (или предвыполнение, если окружение – тело пакета).

Если исключение возбуждено в последовательности операторов окружения, не содержащего обработчика этого исключения, то выполнение последовательности операторов прекращается. Дальнейшие действия зависят от природы окружения:

а) Для тела подпрограммы – то же исключение возбуждается повторно в точке вызова этой подпрограммы, кроме случая, когда она является главной программой. Тогда выполнение главной программы прекращается.

б) Для оператора блока – то же исключение повторно возбуждается непосредственно после оператора блока (т. е. в самом вложенном объемлющем окружении или в операторе принятия, содержащем этот оператор блока).

в) Для тела пакета, являющегося дополнительным элементом описания, – то же исключение возбуждается повторно непосредственно после этого элемента описания (в объемлющем разделе описаний). Если тело этого пакета описано как субмодуль, то исключение возбуждается повторно на месте соответствующего следа тела. Если пакет является библиотечным модулем, то выполнение главной программы прекращается.

г) Для тела задачи – задача становится законченной.

Говорят, что возбуждаемое повторно исключение (как это рассмотрено в подпунктах *а, б* и *в*) *распространяется* либо выполнением подпрограммы, либо выполнением оператора блока, либо предвыполнением тела пакета. В случае тела задачи распространения не происходит. Если окружение является подпрограммой или оператором блока, и если оно содержит зависимые задачи, то распространение исключения происходит только после завершения зависимых задач.

Наконец, если исключение возбуждено в последовательности операторов обработчика исключения, то выполнение этой последовательности операторов прекращается. Последующие действия (включая возможное распространение) зависят от природы окружения и выполняются в соответствии с подпунктами от *а* до *г*.

Пример:

```
function ФАКТОРИАЛ (N: POSITIVE) return FLOAT is
begin
  if N = 1 then
     return 1.0;
  else
     return FLOAT (N) * ФАКТОРИАЛ (N–1);
  end if;
exception
  when NUMERIC_ERROR => return FLOAT'SAFE_LARGE;
end ФАКТОРИАЛ;
```

Если при умножении возбуждается исключение NUMERIC_ERROR, то значение FLOAT'SAFE_LARGE возвращается обработчиком исключения. Это значение будет вызывать возбуждение исключения NUMERIC_ERROR при вычислении выражения в каждом из оставшихся обращений к этой

функции. Таким образом, для больших значений N эта функция всегда будет возвращать значение FLOAT'SAFE_LARGE.

Пример:

```
procedure P is
  ОШИБКА: exception;
  procedure R;
  procedure Q is
  begin
     R;
     . . . -- ошибочная ситуация 2
  exception
     . . .
     when ОШИБКА => -- обработчик E2
  end Q;
  procedure R is
  begin
     . . . -- ошибочная ситуация 3
  end R;
begin
     . . . -- ошибочная ситуация 1
     Q;
     . . .
  exception
     . . .
     when ОШИБКА => -- обработчик E1
  end P;
```

Могут возникнуть следующие случаи.

1. Если исключение ОШИБКА возбуждено при выполнении последовательности операторов внешней процедуры P, то выполнение процедуры P заканчивает обработчик E1, расположенный внутри P.
2. Если исключение ОШИБКА возбуждено при выполнении последовательности операторов процедуры Q, то обработчик E2, расположенный внутри Q, закончит ее выполнение. По окончании выполнения этого обработчика управление будет возвращено в точку вызова процедуры Q.
3. Если исключение ОШИБКА возбуждено в теле процедуры R, вызываемой из процедуры Q, то выполнение процедуры R прекращается, и то же самое исключение возбуждается в теле Q. Затем обработчик E2 заканчивает выполнение процедуры Q, как и в случае 2.

Заметим, что в третьем случае возбуждение исключения в R приводит (косвенно) к передаче управления обработчику, являющемуся частью Q и, следовательно, не вложенному в R. Заметим также, что если бы внутри R был задан обработчик с выбором **others**, то в случае 3, вместо непосредственного завершения R, выполнился бы этот обработчик.

Наконец, если бы исключение ОШИБКА было описано в R, а не в P, то обработчики E1 и E2 не могли бы обеспечивать обработку исключения ОШИБКА, так как этот идентификатор не был бы видимым внутри тел P и Q. В случае 3, однако, это исключение могло бы быть обработано в Q с помощью обработчика с выбором исключения **others**.

Примечание. В языке не определено, что происходит после прекращения выполнения главной программы в результате необработанного исключения.

Предопределенные исключения – это исключения, которые могут распространяться базовыми и предопределенными операциями.

Случай, когда окружением является настраиваемый модуль, уже был учтен в правилах для тел подпрограмм и пакетов, так как последовательность операторов такого окружения не выполняется, а служит шаблоном для конкретизации настройки соответствующей последовательности операторов подпрограммы или пакета.

11.4.2. Исключения, возбуждаемые при предвыполнении описаний

Если исключение возбуждено при предвыполнении раздела описаний данного окружения, то это предвыполнение прекращается. Дальнейшее действие зависит от природы окружения:

а) Для тела подпрограммы – то же исключение повторно возбуждается в точке вызова подпрограммы, кроме случая, когда эта подпрограмма является главной программой, тогда ее выполнение прекращается.

б) Для оператора блока – то же исключение повторно возбуждается непосредственно после оператора блока.

в) Для тела пакета, являющегося дополнительным элементом описания, – то же исключение повторно возбуждается непосредственно после этого элемента описания в объемлющем разделе описаний. Если тело пакета является субмодулем, то исключение возбуждается повторно на месте соответствующего следа тела. Если пакет является библиотечным модулем, то выполнение главной программы прекращается.

г) Для тела задачи – задача становится законченной, а в точке активизации задачи возбуждается исключение TASKING_ERROR, как пояснено в разд. 9.3.

Если исключение возбуждается во время предвыполнения либо описания пакета, либо описания задачи, то это предвыполнение прекращается; дальнейшее действие зависит от природы описания.

д) Для описания пакета или задачи, являющегося элементом описания, то же исключение возбуждается повторно непосредственно после этого элемента описания в объемлющем разделе описаний или в спецификации пакета. Для описания библиотечного пакета – выполнение главной программы прекращается.

Говорят, что возбуждаемое повторно исключение (как рассмотрено выше в подпунктах *а, б, в* и *г*) *распространяется* либо выполнением подпрограммы или оператора блока, либо предвыполнением описания пакета, описания задачи или тела пакета.

Пример исключения в разделе описания оператора блока (случай б):

```
procedure P is
  ...
begin
  declare
     N: INTEGER: = F; - - функция F может возбуждать исключение ОШИБКА
  begin
     ...
  exception
```

```
      when ОШИБКА => -- обработчик E1
    end;
    ...
  exception
    when ОШИБКА => -- обработчик E2
  end P;
    -- Если исключение ОШИБКА возбуждено в описании N,
    -- то оно обрабатывается обработчиком E2.
```

11.5. **Исключения, возбуждаемые при взаимодействии задач**

Исключение может распространяться на взаимодействие задач или на попытку начать взаимодействие одной задачи с другой. Исключение может также распространяться на вызывающую задачу, если оно было возбуждено в процессе рандеву.

Когда задача вызывает вход другой задачи, то в точке этого вызова в вызывающей задаче возбуждается исключение TASKING_ERROR, если вызванная задача закончена до принятия вызова входа или ко времени этого вызова.

Рандеву может иметь аварийное окончание в двух случаях:

а) Если исключение возбуждено в операторе принятия и не обработано во внутреннем окружении. В этом случае выполнение оператора принятия прекращается, и то же исключение повторно возбуждается непосредственно после оператора принятия в вызванной задаче; исключение также распространяется на вызывающую задачу в точку вызова входа.

б) Если задача, содержащая оператор принятия, закончена аварийно в результате выполнения оператора прекращения. В этом случае исключение TASKING_ERROR возбуждается в вызывающей задаче в точке вызова входа.

С другой стороны, если задача, вызывающая вход, аварийно прекращает свое выполнение (в результате выполнения оператора прекращения), то в вызванной задаче исключение не возбуждается. Если рандеву еще не началось, то вызов входа аннулируется. Если же рандеву началось, то оно заканчивается нормально, и это никак не влияет на вызванную задачу.

11.6. **Исключения и оптимизация**

В данном разделе описаны условия, при которых в реализации можно выполнять те или иные действия раньше или позже, чем это определено правилами языка.

В целом, если правила языка задают порядок некоторых действий (*канонический порядок*), реализация может использовать альтернативный порядок при гарантиях, что такое переупорядочивание не скажется на результате выполнения программы. В частности, если при выполнении программы в каноническом порядке не возбуждается никакое исключение, то также никакие исключения не должны возбуждаться при выполнении переупорядоченной программы. С другой стороны, если порядок некоторых действий не определен языком, то реализация может использовать любой порядок. (Например, аргументы предопределенной операции могут вычисляться в любом порядке, так как правила из разд. 4.5 не требуют определенного порядка выполнения.)

Реализации предоставляется дополнительная свобода для переупорядочивания действий, включающих предопределенные или базовые операции, за исключением присваивания. Эта свобода предоставляется даже в том случае, если при выполнении предопределенных операций может распространяться (предопределенное) исключение с учетом следующих правил:

а) С целью установления, одинаков ли результат выполнения некоторых действий в каноническом или в альтернативном порядках, можно предположить, что ни одна из вызванных этими действиями предопределенных операций не распространяет (предопределенное) исключение, и при этом выполняются два следующих требования к реализации альтернативного порядка: во-первых, операция не должна вызываться в альтернативном порядке, если она не вызывается в каноническом порядке; во-вторых, самое внутреннее объемлющее окружение или оператор принятия для каждой операции должны быть одинаковы для канонического и альтернативного порядков с теми же самыми обработчиками исключений.

б) Связь знаков операций с операндами в выражении определена синтаксисом. Однако для последовательности предопределенных операций с одним и тем же приоритетом (при отсутствии скобок, вводящих особые связи) эти связи могут быть изменены (и это допускается) при выполнении следующего требования: результат целого типа должен быть эквивалентен результату вычислений в каноническом порядке слева направо; результат вещественного типа должен принадлежать модельному интервалу результата, полученного после выполнения в каноническом порядке слева направо (см. 4.5.7). Такое переупорядочивание допустимо даже тогда, когда может быть устранено некоторое исключение или вставлено предопределенное исключение.

Также дополнительная свобода предоставляется реализации при вычислении простых числовых выражений. При выполнении предопределенных операций в реализации допускается использование операций над типами, которые имеют более широкий диапазон результата, чем базовый тип операндов, при условии, что это приводит к точному результату (или результату с заданной точностью для вещественного типа), даже если промежуточные результаты выходят за границы базового типа. В таком случае нет необходимости возбуждать исключение NUMERIC_ERROR. В частности, если числовое выражение является операндом предопределенной операции отношения и может быть получен правильный результат типа BOOLEAN, то в процессе вычисления можно не возбуждать исключение NUMERIC_ERROR.

Нет необходимости в выполнении предопределенной операции, если ее единственным возможным результатом является распространение предопределенного исключения или если изменение последовательности операций по описанным выше правилам приводит к ее безрезультатному выполнению.

Примечание. Правило *б* применимо к предопределенным операциям, но не применимо к формам управления с промежуточной проверкой.

Выражение СКОРОСТЬ < 300_000.0 может быть заменено на TRUE, если значение 300_000.0 находится вне границ базового типа для СКОРОСТЬ, даже если неявное преобразование этого числового литерала может возбудить исключение NUMERIC_ERROR.

Пример:

```
declare
  N: INTEGER;
begin
  N: = 0;                    - - (1)
  for J in 1. .10 loop
     N: = N + J ** A(K) ; - - A и K являются
                          - - глобальными переменными
  end loop;
  PUT(N);
exception
  when others => PUT ("ОБНАРУЖЕННАЯ_ОШИБКА") ; PUT(N);
end;
```

Вычисление A(K) может быть выполнено до цикла и, возможно, непосредственно перед оператором присваивания (1), даже если в нем может возбудиться исключение. Следовательно, внутри обработчика исключения значение N будет либо неопределенным, либо результатом последнего присваивания. С другой стороны, вычисление A(K) не может быть выполнено до **begin**, поскольку в этом случае исключение будет обрабатываться другим обработчиком. По этой причине инициализация N в описании будет исключать возможность наличия неопределенного начального значения N в обработчике.

11.7. **Подавление проверок**

Присутствие прагмы SUPPRESS позволяет реализации опускать некоторые проверки во время выполнения программы. Эта прагма имеет следующий вид:

pragma SUPPRESS (идентификатор [, [ON =>] имя]) ;

Проверка, указанная идентификатором, может быть опущена. Имя (если оно присутствует) должно быть простым или расширенным именем и должно обозначать объект, тип или подтип, задачный модуль или настраиваемый модуль; оно также может быть именем подпрограммы, в этом случае имя обозначает все видимые совмещенные подпрограммы.

Прагма SUPPRESS допустима непосредственно в разделе описаний или непосредственно в спецификации пакета. Во втором случае допустимо представление прагмы только с именем, которое обозначает понятие (или несколько совмещенных подпрограмм), описанное непосредственно в спецификации пакета. Действие прагмы распространяется от месторасположения прагмы до конца зоны описания, связанной с самым внутренним объемлющим оператором блока или программным модулем. Если прагма задана в спецификации пакета, то ее действие распространяется до конца области действия именованного понятия.

Если в прагму включено имя, то возможность подавления проверки в дальнейшем ограничена: прагма действует только для операций над объектом с этим именем или над всеми объектами базового типа для указанного

в прагме имени типа или подтипа; для вызовов подпрограмм с этим именем; для активизации задач указанного именем задачного типа; для конкретизаций указанного настраиваемого модуля.

Следующие проверки соответствуют ситуациям, в которых может быть возбуждено исключение CONSTRAINT_ERROR. В этих проверках имя (если оно указано) должно обозначать объект или тип (в скобках приведен русский перевод).

ACCESS_CHECK (ПРОВЕРКА_ССЫЛКИ) Проверяется именуемый компонент, индексируемый компонент, отрезок или атрибут объекта, указанный ссылочным значением, на неравенство значению **null** этого ссылочного значения.

DISCRIMINANT_CHECK (ПРОВЕРКА_ДИСКРИМИНАНТА) Проверяется, что дискриминант составного значения удовлетворяет ограничению дискриминанта. Также при ссылке на компоненты записи проверяется их существование для текущих значений дискриминанта.

INDEX_CHECK (ПРОВЕРКА_ИНДЕКСА) Проверяется, что границы значений индексов массива равны соответствующим границам ограничения индекса. Также при ссылке на компонент массива по каждой размерности проверяется, что данное значение индекса находится в диапазоне, определенном границами индекса массива; при ссылке на отрезок массива проверяется, что заданный дискретный диапазон совместим с диапазоном, определенным границами индексов массива.

LENGTH_CHECK (ПРОВЕРКА_ДЛИНЫ) Проверяется, что каждому компоненту массива соответствует подходящий компонент при выполнении присваивания массиву, преобразовании типа и выполнении логических операций над массивами логических компонентов.

RANGE_CHECK (ПРОВЕРКА_ДИАПАЗОНА) Проверяется, что некоторое значение удовлетворяет ограничению диапазона. Также при предвыполнении указания подтипа проверяется совместимость ограничения (если оно имеется) с обозначением типа. Для агрегата проверяется принадлежность индекса или дискриминанта соответствующему подтипу. Наконец, осуществляются проверки любых ограничений, создаваемых при конкретизации настройки.

Следующие проверки соответствуют ситуациям, в которых возбуждается исключение NUMERIC_ERROR. Допустимыми именами в соответствующих прагмах являются имена числовых типов.

DIVISION_CHECK (ПРОВЕРКА_ДЕЛЕНИЯ) Проверяется, что второй операнд операций /, **rem** и **mod** не равен нулю.

OVERFLOW_CHECK (ПРОВЕРКА_ПЕРЕПОЛНЕНИЯ) Проверяется, что в результате выполнения числовой операции не возникает переполнения.

Следующие проверки соответствуют ситуациям, в которых возбуждается исключение PROGRAM_ERROR. Допустимыми именами в соответствующих прагмах являются имена задачных модулей, настраиваемых модулей или подпрограмм.

ELABORATION_CHECK (ПРОВЕРКА_ПРЕДВЫПОЛНЕНИЯ) Когда вызывается подпрограмма, выполняется активизация задачи или предвыполняется конкретизация настройки, то проверяется, что тело соответствующего программного модуля уже предвыполнено.

Следующие проверки соответствуют ситуациям, в которых возбуждается исключение STORAGE_ERROR. Допустимыми именами в соответствующих прагмах являются имена, обозначающие ссылочные типы, задачные модули или подпрограммы.

STORAGE_CHECK (ПРОВЕРКА_ПАМЯТИ) Проверяется, что выполнение генератора не потребует объем памяти, больший необходимого для набора, или что требуемый под задачу или для подпрограммы объем памяти достаточен.

Если возникает ошибочная ситуация в отсутствии проверок во время выполнения программы, то программа считается ошибочной (результаты выполнения не определяются в языке).

Примеры:

pragma SUPPRESS (RANGE_CHECK);

pragma SUPPRESS (INDEX_CHECK, ON => ТАБЛИЦА);

Примечание. Для некоторых реализаций может оказаться невозможным или слишком дорогим подавление некоторых проверок, тогда соответствующая прагма SUPPRESS может быть проигнорирована. Следовательно, наличие такой прагмы внутри данного модуля не гарантирует, что соответствующее исключение не будет возбуждено; эти исключения также могут распространяться вызванными модулями.

12. НАСТРАИВАЕМЫЕ МОДУЛИ

Настраиваемый модуль – это программный модуль, являющийся настраиваемой подпрограммой или настраиваемым пакетом, это также *шаблон* с параметрами или без них, по которому могут быть получены соответствующие (не настраиваемые) подпрограммы или пакеты. Итоговые программные модули называются *экземплярами* исходного настраиваемого модуля.

Настраиваемый модуль задается описанием настройки, которое имеет раздел формальных параметров настройки, описывающий эти параметры. Конкретный экземпляр настраиваемого модуля получается в результате конкретизации настройки путем сопоставления формальным параметрам фактических. Экземпляр настраиваемой подпрограммы – это подпрограмма. Экземпляр настраиваемого пакета – пакет.

Как шаблоны, настраиваемые модули не обладают свойствами, характерными для их ненастраиваемых аналогов. Например, настраиваемая подпрограмма может быть конкретизирована, но не может быть вызвана. В отличие от нее экземпляр настраиваемой подпрограммы может быть вызван, но не может использоваться для изготовления других экземпляров.

12.1. Описание настройки

Описание настройки задает настраиваемый модуль: настраиваемую подпрограмму или настраиваемый пакет. Описание настройки включает раздел

формальных параметров настройки, в котором описываются ее формальные параметры. Формальный параметр настройки может быть объектом; кроме того (в отличие от параметра подпрограммы), он может быть типом или подпрограммой.

описание_настройки : : = спецификация_настройки;
спецификация_настройки : : =
раздел_формальных_параметров_настройки
спецификация_подпрограммы
| раздел_формальных_параметров_настройки
спецификация_пакета
раздел_формальных_параметров_настройки : : =
generic {описание_параметра_настройки}
описание_параметра_настройки : : =
список_идентификаторов :
[**in** [**out**]] обозначение_типа [: = выражение] ;
| **type** идентификатор **is** определение_настраиваемого_типа;
| описание_личного_типа
| **with** спецификация_подпрограммы [**is** имя] ;
| **with** спецификация_подпрограммы [**is** <>] ;
определение_настраиваемого_типа : : =
(<>) | **range** <>| **digits** <> | **delta** <>
| определение_индексируемого_типа
| определение_ссылочного_типа

Для ссылки на соответствующие формальные параметры настройки используются такие термины: формальный объект настройки (или, короче, *формальный объект*), формальный тип настройки (или, короче, *формальный тип*) и формальная подпрограмма настройки (или, короче, *формальная подпрограмма*).

В разделе формальных параметров настройки указание подтипа допустимо только в виде обозначения типа (т. е. такое указание подтипа не должно содержать явного ограничения). Обозначение настраиваемой подпрограммы должно быть задано идентификатором.

Имя программного модуля, являющегося настраиваемым модулем, вне его спецификации и тела обозначает этот настраиваемый модуль. В отличие от этого, в зоне описания, связанной с настраиваемой подпрограммой, имя такого программного модуля обозначает подпрограмму, полученную при текущей конкретизации настраиваемого модуля. Аналогично, в зоне описания, связанной с настраиваемым пакетом, имя этого программного модуля обозначает пакет, полученный при текущей конкретизации.

Предвыполнение описания настройки не имеет другого эффекта.

Примеры разделов формальных параметров:

```
generic - - без параметров
generic
  РАЗМЕР: NATURAL; - - формальный объект
generic
  ДЛИНА: INTEGER: = 200; - - формальный объект с выражением по умолчанию
```

```
  ПЛОЩАДЬ: INTEGER: = ДЛИНА * ДЛИНА;
                                  - - формальный объект с выражением по умолчанию
generic
  type ЭЛЕМЕНТ is private; - - формальный тип
  type ИНДЕКС is (<>); - - формальный тип
  type РЯД is array (ИНДЕКС range <>) of ЭЛЕМЕНТ; - - формальный тип
  with function "<" (X, Y: ЭЛЕМЕНТ) return BOOLEAN;
                                                  формальная подпрограмма
```

Примеры описаний настройки с настраиваемыми подпрограммами:

```
generic
  type ЭЛЕМ is private;
procedure ЗАМЕНА (A, B: in out ЭЛЕМ);
generic
  type ЭЛЕМЕНТ is private;
  with function "*" (A, B: ЭЛЕМЕНТ) return ЭЛЕМЕНТ is < >;
function ВОЗВ_В_КВАДРАТ (X: ЭЛЕМЕНТ) return ЭЛЕМЕНТ;
```

Пример описания настройки с настраиваемым пакетом:

```
generic
  type ЭЛЕМЕНТ is private;
  type ВЕКТОР is array (POSITIVE range < >) of ЭЛЕМЕНТ;
  with function СУММА (X, Y: ЭЛЕМЕНТ) return ЭЛЕМЕНТ;
package НАД_ВЕКТОРАМИ is
  function СУММА (A, B: ВЕКТОР) return ВЕКТОР;
  function СИГМА (A: ВЕКТОР) return ЭЛЕМЕНТ;
  ОШИБОЧНАЯ_ДЛИНА: exception;
end;
```

Примечание. Внутри тела настраиваемой подпрограммы ее имя рассматривается как имя подпрограммы. Следовательно, это имя может быть совмещено, а также может появиться в рекурсивном вызове текущей конкретизации. По этой же причине его нельзя использовать после зарезервированного слова new в (рекурсивной) конкретизации настройки.

Выражение, которое находится в разделе формальных параметров настройки, – это выражение по умолчанию для формального параметра вида in, либо составная часть имени входа, заданного как имя по умолчанию для формальной подпрограммы, либо выражение по умолчанию для параметра формальной подпрограммы. В первых двух случаях значение этого выражения вычисляется только в тех конкретизациях, в которых используется соответствующее умолчание. В третьем случае значение выражения вычисляется только в вызовах формальных подпрограмм, использующих такое умолчание. (К любому имени, используемому в выражении по умолчанию, применяются обычные правила видимости: обозначенные этим именем понятия должны быть видимы в том месте, где стоит выражение.)

Ни формальные параметры настройки, ни их атрибуты в качестве частей статических выражений не допустимы (см. 4.9).

12.1.1. Формальные объекты настройки

Первая форма описания формального параметра настройки задает формальные объекты настройки. Тип формального объекта настройки – это базовый тип обозначения типа, данного в описании формального параметра настройки. Описание формального параметра настройки с несколькими идентификаторами эквивалентно последовательности единичных описаний, как поясняется в разд. 3.2.

Формальный объект настройки имеет вид in или in out. При отсутствии в описании формального параметра настройки явного указания вида подра-

зумевается вид in. Если в описании формального параметра настройки задано выражение, то оно является *выражением по умолчанию* для этого формального параметра. Выражение по умолчанию допустимо только для параметров вида in (указанного либо явно, либо неявно). Тип выражения по умолчанию должен быть таким же, как и у соответствующего формального параметра настройки.

Формальный объект настройки вида in – это константа, значение которой является копией значения сопоставленного ему фактического параметра конкретизации настройки, как описано в разд. 12.3. Тип формального объекта настройки вида in не должен быть лимитируемым типом; подтип такого формального объекта настройки – это подтип в обозначении типа, данного в описании параметра настройки.

Формальный объект настройки вида in out – это переменная, обозначающая объект, задаваемый в конкретизации настройки фактическим параметром настройки, как описано в разд. 12.3. Ограничения, применяемые к формальному объекту настройки, те же, что и для соответствующего фактического параметра.

Примечание. Ограничения, применяемые к формальному объекту настройки вида in out, те же, что и для соответствующего фактического параметра (а не те, которые связаны с обозначением типа из описания параметра настройки). Во избежание путаницы рекомендуется, когда это возможно, использовать в описании такого формального объекта имя базового типа. Если, однако, базовый тип анонимен, то рекомендуется использовать имя подтипа, определенного в описании базового типа.

12.1.2. Формальные типы настройки

Описание параметра настройки, включающее определение настраиваемого типа или описание личного типа, задает формальный тип настройки. Формальный тип настройки обозначает подтип, заданный соответствующим фактическим параметром в конкретизации настройки, как описано в разд. 12.3 г. В настраиваемом модуле формальный тип настройки рассматривается как некоторый уникальный тип, отличный от всех остальных (формальных или нет) типов. Форма ограничения, применимого к формальному типу в указании подтипа, зависит от класса типа, как и для типов, не являющихся формальными.

В описании формального (ограниченного) индексируемого типа настройки в качестве формы дискретного диапазона допустимо только обозначение типа.

Раздел дискриминантов формального личного типа настройки не должен включать выражение по умолчанию для дискриминанта. (Следовательно, переменная, заданная описанием объекта, должна быть ограничена, если ее тип – это формальный тип настройки с дискриминантами.)

В описании и теле настраиваемого модуля операции, которые можно выполнять над значениями формального типа настройки (кроме дополнительных операций, заданных формальными подпрограммами настройки), определяются описанием параметра настройки для этого формального типа:

а) Для описания личного типа разрешены операции, определенные в разд. 7.4.2 (в частности, для личного, но не лимитируемого, типа – присваивание, равенство и неравенство).

б) Для определения индексируемого типа разрешены операции, определенные в разд. 3.6.2 (например, они включают формирование индексируемых компонентов и отрезков).

в) Для определения ссылочного типа разрешены операции, определенные в разд. 3.8.2 (например, могут быть использованы генераторы).

Четыре формы определения настраиваемого типа, в которых содержится *бокс* (т. е. составной ограничитель < >), соответствуют следующим основным формам скалярного типа:

г) Дискретные типы: (< >).

Разрешенные операции – общие для перечислимых и целых типов; они определены в разд. 3.5.5.

д) Целые типы: **range** < >.

Разрешенные операции над целыми типами определены в разд. 3.5.5.

е) Плавающие типы: **digits** < >.

Разрешенные операции определены в разд. 3.5.8.

ж) Фиксированные типы: **delta** < >.

Разрешенные операции определены в разд. 3.5.10.

Во всех случаях, от *а* до *е*, каждая операция, неявно связанная с формальным типом (т. е. *отличная от операции, заданной формальной подпрограммой*), считается неявно описанной в месте описания формального типа. Это же относится и к формальному фиксированному типу, исключая мультипликативные операции, которые возвращают результат *универсального_фиксированного* типа (см. 4.5.5), так как эти специальные операции описаны в пакете STANDARD.

При конкретизации настройки каждая из этих операций – это соответствующая базовая операция или предопределенная операция для сопоставленного фактического типа. Для операции это правило сохраняется даже в случае переопределения ее для фактического типа или некоторого его родительского типа.

Примеры формальных типов настройки:

```
type ЭЛЕМЕНТ is private;
type БУФЕР (ДЛИНА: NATURAL) is limited private;
type ПЕРЕЧИСЛ is (< >);
type ЦЕЛ is range < >;
type УГОЛ is delta < >;
type МАССА is digits < >;
type ТАБЛИЦА is array (ПЕРЕЧИСЛ) of ЭЛЕМЕНТ;
```

Пример раздела формальных параметров настройки с описанием формального целого типа:

```
generic
  type РЯД is range < >;
  ПЕРВЫЙ: РЯД: = РЯД'FIRST;
  ВТОРОЙ: РЯД: = ПЕРВЫЙ + 1; - - операция "+" для типа РЯД
```

12.1.3. Формальные подпрограммы настройки

Описание параметра настройки, включающее спецификацию подпрограммы, описывает формальную подпрограмму настройки.

В описании формальной подпрограммы настройки могут встречаться две формы умолчания. В них после спецификации подпрограммы следует зарезервированное слово is и либо бокс, либо имя подпрограммы или входа. Правила сопоставления для таких умолчаний описаны в разд. 12.3.6.

Формальная подпрограмма настройки обозначает подпрограмму, литерал перечисления или вход, заданный соответствующим фактическим параметром настройки в конкретизации настройки, как описано в разд. 12.3 е.

Примеры формальных подпрограмм настройки:

```
with function УВЕЛИЧИТЬ (X: INTEGER) return INTEGER;
with function СУММА (X, Y: ЭЛЕМЕНТ) return ЭЛЕМЕНТ;
with function "+" (X, Y: ЭЛЕМЕНТ) return ЭЛЕМЕНТ is < >;
with function ОБРАЗ (X: ПЕРЕЧИСЛ) return STRING is
                ПЕРЕЧИСЛ'IMAGE;
with procedure ОБНОВЛЕНИЕ is ОБНОВЛЕНИЕ_ПО_УМОЛЧАНИЮ;
```

Примечание. Ограничения на параметр формальной подпрограммы те же, что у соответствующего параметра в спецификации сопоставленной фактической подпрограммы (а не те, которые вводятся соответствующим обозначением типа в спецификации формальной подпрограммы). Это же относится и к результату функции. Во избежание путаницы рекомендуется везде, где можно, в описании формальной подпрограммы использовать имя базового типа, а не имя подтипа. Если, однако, базовый тип анонимен, то рекомендуется использовать имя подтипа, определенное в описании типа.

Тип, заданный для формального параметра формальной подпрограммы настройки, может быть любым видимым типом, включая формальный тип настройки из того же раздела формальных параметров настройки.

12.2. Настраиваемое тело

Тело настраиваемой подпрограммы или настраиваемого пакета (настраиваемое тело) является шаблоном для тел соответствующих подпрограмм или пакетов, получаемых конкретизацией настройки. Синтаксис настраиваемого тела идентичен обычному телу.

Для каждого описания настраиваемой подпрограммы должно быть соответствующее тело.

Предвыполнение настраиваемого тела не имеет другого эффекта, кроме установления того, что тело, начиная с этого момента, может быть использовано в качестве шаблона для получения соответствующих экземпляров.

Пример настраиваемого тела процедуры:

```
procedure ЗАМЕНА (A, B: in out ЭЛЕМ) is - - см. пример в 12.1
  T: ЭЛЕМ; - - формальный тип настройки
begin
  T: = A;
  A: = B;
  B: = T;
end ЗАМЕНА;
```

Пример тела настраиваемой функции:

```
function ВОЗВ_В_КВАДРАТ (X: ЭЛЕМЕНТ) return ЭЛЕМЕНТ is
begin                          - - см. пример в 12.1
  return X * X;                - - формальная операция "*"
end;
```

Пример тела настраиваемого пакета:

```
package body НАД_ВЕКТОРАМИ is - - см. пример в 12.1
```

```
  function СУММА (A, B: ВЕКТОР) return ВЕКТОР is
     РЕЗУЛЬТАТ: ВЕКТОР (A'RANGE); - - формальный тип ВЕКТОР
     СМЕЩЕНИЕ: constant INTEGER := B'FIRST – A'FIRST;
  begin
     if A'LENGTH / = B'LENGTH then
       raise ОШИБОЧНАЯ_ДЛИНА;
     end if;
     for K in A'RANGE loop
        РЕЗУЛЬТАТ (К): = СУММА (A(К), B(К + СМЕЩЕНИЕ));
                                          - - формальная функция СУММА

     end loop;
     return РЕЗУЛЬТАТ;
  end;
  function СИГМА (A: ВЕКТОР) return ЭЛЕМЕНТ is
     ИТОГ: ЭЛЕМЕНТ: = A (A'FIRST); - - формальный тип ЭЛЕМЕНТ
  begin
     for K in A'FIRST + 1. .A'LAST loop
        ИТОГ: = СУММА (ИТОГ, A (К)); - - формальная функция СУММА
     end loop;
     return ИТОГ;
  end;
end;
```

12.3. **Конкретизация настройки**

Экземпляр настраиваемого модуля описывается конкретизацией настройки.

конкретизация_настройки : : =
  package идентификатор **is new** имя_*настраиваемого_пакета*
    [раздел_фактических_параметров_настройки] ;
  | **procedure** идентификатор **is new**
    имя_*настраиваемой_процедуры*
    [раздел_фактических_параметров_настройки] ;
  | **function** обозначение **is new** имя_*настраиваемой_функции*
    [раздел_фактических_параметров_настройки] ;
раздел_фактических_параметров_настройки : : =
  (сопоставление_параметров_настройки
  {, сопоставление_параметров_настройки})
сопоставление_параметров_настройки : : =
  [формальный_параметр_настройки =>]
  фактический_параметр_настройки
формальный_параметр_настройки : : =
  простое_имя_*параметра* | знак_операции
фактический_параметр_настройки : : = выражение
  | имя_*переменной* | имя_*подпрограммы*
  | имя_*входа* | обозначение_типа

Для каждого формального параметра настройки должен быть задан явный фактический параметр настройки, кроме случая, когда соответствующим описанием параметра настройки задана форма умолчания. Сопоставление параметров настройки может быть либо позиционным, либо именован-

ным (как в вызове подпрограммы (см. 6.4)). Если две или несколько формальных подпрограмм имеют одно и то же обозначение, то для соответствующих параметров настройки именованные сопоставления недопустимы.

Каждый фактический параметр настройки должен быть *сопоставлен* с соответствующим формальным параметром. Выражение может быть сопоставлено с формальным объектом вида **in**; имя переменной может быть сопоставлено с формальным объектом вида **in out**; имя подпрограммы или имя входа может сопоставляться с формальной подпрограммой; обозначение типа может сопоставляться с формальным типом. Детальные правила, определяющие единственные допустимые сопоставления, даны в разд. 12.3.1–12.3.6.

Экземпляр – это копия настраиваемого модуля без его раздела формальных параметров настройки; таким образом, экземпляр настраиваемого пакета – пакет, настраиваемой процедуры – процедура, настраиваемой функции – функция. Для каждого вхождения обозначающего данное понятие имени в настраиваемый модуль следующий список определяет, какое понятие соответствует этому имени в экземпляре.

а) Имя обозначает этот настраиваемый модуль; соответствующее вхождение обозначает экземпляр.

б) Имя обозначает формальный объект настройки вида **in**; соответствующее в экземпляре имя обозначает константу, значение которой – копия значения сопоставленного фактического параметра настройки.

в) Имя обозначает формальный объект настройки вида **in out**; соответствующее в экземпляре имя обозначает переменную, указанную сопоставленным фактическим параметром настройки.

г) Имя обозначает формальный тип настройки; соответствующее в экземпляре имя обозначает подтип, указанный сопоставленным фактическим параметром настройки (фактическим подтипом).

д) Имя обозначает дискриминант формального типа настройки; соответствующее в экземпляре имя обозначает соответствующий дискриминант (он должен быть один) фактического типа, сопоставленного формальному типу настройки.

е) Имя обозначает формальную подпрограмму настройки; соответствующее в экземпляре имя обозначает подпрограмму, литерал перечисления или вход, указанный сопоставленным фактическим параметром настройки (фактической подпрограммой).

ж) Имя обозначает формальный параметр формальной подпрограммы настройки; соответствующее в экземпляре имя обозначает соответствующий формальный параметр фактической подпрограммы, соответствующей формальной подпрограмме.

з) Имя обозначает локальное понятие, описанное в настраиваемом модуле; соответствующее в экземпляре имя обозначает понятие, описанное соответствующим локальным описанием в экземпляре.

и) Имя обозначает глобальное понятие, описанное вне настраиваемого модуля; соответствующее в экземпляре имя обозначает это же глобальное понятие.

Те же правила справедливы для знаков операций и базовых операций, в частности, для формальных операций верно правило *е*, для локальных операций – правило *з* и для операций над глобальными типами – правило *и*. Кроме того, если в настраиваемом модуле используется предопределенная операция или базовая операция над формальным типом, то в экземпляре используется предопределенная операция, соответствующая фактическому типу, сопоставленного формальному.

Эти же правила применяются к обозначению типа и выражению (по умолчанию) из раздела формальных параметров настройки настраиваемого модуля.

При предвыполнении конкретизации настройки осуществляются следующие действия: сначала вычисляется каждое выражение, заданное в качестве явного фактического параметра настройки, и каждое выражение, входящее как составная часть в имя переменной или входа, заданное в качестве явного фактического параметра настройки; в языке не определен порядок вычисления этих выражений. Затем вычисляются выражения или имена по умолчанию для тех параметров, для которых опущены сопоставления (если они есть) параметров настройки; эти вычисления производятся в порядке следования описаний формальных параметров настройки. Наконец, предвыполняется неявно сгенерированный экземпляр. Предвыполнение конкретизации настройки может также вызывать проверки некоторых ограничений, как описано ниже.

Рекурсивная конкретизация настройки недопустима в следующем смысле: если данный настраиваемый модуль включает конкретизацию другого настраиваемого модуля, то экземпляр, сгенерированный этой конкретизацией, не должен включать экземпляр первого настраиваемого модуля (независимо от того, генерируется ли этот экземпляр непосредственно или косвенно через промежуточную конкретизацию).

Примеры конкретизации настройки (см. 12.1):

```
procedure ОБМЕН is new ЗАМЕНА (ЭЛЕМ => INTEGER);
procedure ОБМЕН is new ЗАМЕНА (CHARACTER);
  -- совмещение идентификатора ОБМЕН
function КВАДРАТ is new ВОЗВ_В_КВАДРАТ (INTEGER);
  -- по умолчанию используется "*" над INTEGER
function КВАДРАТ is new ВОЗВ_В_КВАДРАТ (ЭЛЕМЕНТ => МАТРИЦА,
                                        "*" => ПРОИЗВЕДЕНИЕ_МАТРИЦ);
function КВАДРАТ is new ВОЗВ_В_КВАДРАТ (МАТРИЦА,
                                                ПРОИЗВЕДЕНИЕ_МАТРИЦ);
  -- что эквивалентно предыдущему
package ЦЕЛ_ВЕКТОРЫ is new НАД_ВЕКТОРАМИ (INTEGER, ТАБЛИЦА, "+");
```

Примеры использования конкретизированных модулей

```
ОБМЕН (A, B);
A: = КВАДРАТ (A);
T: ТАБЛИЦА (1. .5): = (10, 20, 30, 40, 50);
K: INTEGER: = ЦЕЛ_ВЕКТОРЫ. СИГМА (T); -- 150 (см. 12.2)
use ЦЕЛ_ВЕКТОРЫ;
M: INTEGER: = СИГМА (T); -- 150
```

Примечание. Опускать параметр настройки допускается только тогда, когда для него существует умолчание. Если использованы выражение по умолчанию или (отличные от простых) имена по умолчанию, то они вычисляются в том порядке, в котором описаны соответствующие формальные параметры настройки.

Если две совмещенные подпрограммы описаны в спецификации настраиваемого пакета и различаются только (формальным) типом параметров и результата, то существуют правильные конкретизации, для которых все вызовы этих подпрограмм вне экземпляра будут неоднозначными. Например:

```
generic
  type A is (< >);
  type B is private;
package G is
  function СЛЕДУЮЩИЙ (X:A) return A;
  function СЛЕДУЮЩИЙ (X:B) return B;
end;
package P is new G (A => BOOLEAN, B => BOOLEAN);
  - - все вызовы P. СЛЕДУЮЩИЙ неоднозначны.
```

12.3.1. Правила сопоставления для формальных объектов

Формальному параметру настройки вида **in** данного типа сопоставляется выражение этого же типа. Если настраиваемый модуль имеет формальный объект настройки вида **in**, то проверяется принадлежность значения выражения подтипу, заданному обозначением типа, как и для явного описания константы (см. 3.2.1). При отрицательном результате проверки этого возбуждается исключение CONSTRAINT_ERROR.

Формальному параметру настройки вида **in out** данного типа сопоставляется имя переменной этого же типа. Переменная не должна быть формальным параметром вида **out** или его подкомпонентом. Имя должно обозначать такую переменную, для которой допустимо переименование (см. 8.5).

Примечание. Тип фактического параметра настройки вида **in** не должен быть лимитируемым типом. Ограничения формального параметра настройки вида **in out** являются ограничениями соответствующего фактического параметра настройки (см. 12.1.1).

12.3.2. Правила сопоставления для формальных личных типов

Формальный личный тип настройки сопоставляется с типом или подтипом (фактическим подтипом), удовлетворяющим следующим условиям:

• Если формальный тип не является лимитируемым, то фактический тип не должен быть лимитируемым. (С другой стороны, если формальный тип является лимитируемым, то соответствующий фактический тип может быть лимитируемым и нелимитируемым.)

• Если формальный тип имеет раздел дискриминантов, то фактический тип должен быть типом с таким же числом дискриминантов; тип дискриминанта в данной позиции в разделе дискриминантов фактического типа должен совпадать с типом дискриминанта в той же позиции раздела дискриминантов формального типа; фактический подтип должен быть неограниченным. (С другой стороны, если формальный тип на имеет дискриминантов, для фактического типа дискриминанты допустимы.)

Ниже рассматривается вхождение имени формального типа в том месте, где оно использовано как указание неограниченного подтипа. Фактический подтип не должен быть неограниченным индексируемым типом или неограниченным типом с дискриминантами, если любое вхождение находится на месте, где для индексируемого типа или типа с дискриминантами требуется либо ограничение, либо выражения по умолчанию для дискриминантов (см. 3.6.1 и 3.7.2). Такое же требование предъявляется ко всем вхождениям имени подтипа формального типа, а также к вхождениям имени любого типа или подтипа, производных (непосредственно или косвенно) от этого формального типа.

Если настраиваемый модуль имеет формальный личный тип с дискриминантами, то при предвыполнении соответствующей конкретизации настройки проверяется совпадение подтипа каждого дискриминанта фактического типа и подтипа соответствующего дискриминанта формального типа. При несовпадении таких подтипов возбуждается исключение CONSTRAINT_ERROR.

12.3.3. Правила сопоставления для формальных скалярных типов

Формальному типу настройки, определенному символами (<>), сопоставляется любой дискретный подтип (т. е. любой перечислимый или целый подтип). Формальному типу настройки, определенному символами **range** <>, сопоставляется любой целый подтип. Формальному типу настройки, определенному символами **digits** < >, сопоставляется любой плавающий подтип. Формальному типу настройки, определенному символами **delta** < >, сопоставляется любой фиксированный подтип. Никакие другие сопоставления для этих формальных типов настройки невозможны.

12.3.4. Правила сопоставления для формальных индексируемых типов

Формальному индексируемому типу сопоставляется фактический индексируемый подтип, удовлетворяющий следующим условиям.

• Формальный и фактический индексируемые типы должны иметь одинаковые размерности; формальный тип и фактический подтип должны быть либо оба ограниченными, либо оба неограниченными.

• Для каждой позиции индекса тип индекса фактического индексируемого типа должен совпадать с типом индекса формального индексируемого типа.

• Типы компонентов фактического и формального индексируемых типов должны быть одинаковыми. Если тип компонента отличен от скалярного, то подтипы компонентов фактического и формального индексируемых типов должны быть либо оба ограниченными, либо оба неограниченными.

Если настраиваемый модуль имеет формальный индексируемый тип, то при предвыполнении соответствующей конкретизации проверяется совпадение ограничений (если они есть) на тип компонента фактического индексируемого типа с ограничениями для формального индексируемого типа; для любой данной позиции индекса индексируемых подтипов или дискретных диапазонов проверяется совпадение границ.

При несовпадениях возбуждается исключение CONSTRAINT_ERROR.

Пример:

```
- - задание настраиваемого пакета
generic
  type ЭЛЕМЕНТ is private;
  type ИНДЕКС is (< >);
  type ВЕКТОР is array (ИНДЕКС range < >) of ЭЛЕМЕНТ;
  type ТАБЛИЦА is array (ИНДЕКС) of ЭЛЕМЕНТ;
package P is
end;
- - даны типы
type СМЕСЬ is array (ЦВЕТ range < >) of BOOLEAN;
type ВЕРСИЯ is array (ЦВЕТ) of BOOLEAN;
- - теперь тип СМЕСЬ может быть сопоставлен типу ВЕКТОР, а ВЕРСИЯ – типу
- - ТАБЛИЦА
package R is new P (ЭЛЕМЕНТ => BOOLEAN, ИНДЕКС => ЦВЕТ,
      ВЕКТОР => СМЕСЬ, ТАБЛИЦА => ВЕРСИЯ);
- - Заметим, что СМЕСЬ не может быть сопоставлена с ТАБЛИЦА, а ВЕРСИЯ – с
- - ВЕКТОР.
```

Примечание. Если тип любого индекса или тип компонента формального индексируемого типа сам является формальным типом, то по приведенным правилам в экземпляре его имя обозначает соответствующий фактический подтип (см. 12.3 г).

12.3.5. Правила сопоставления для формальных ссылочных типов

Формальному ссылочному типу сопоставляется фактический ссылочный подтип, если тип указываемых объектов для формального и фактического типов один и тот же. Если указываемый тип отличен от скалярного, то указанные подтипы должны быть либо оба ограниченными, либо оба неограниченными.

Если настраиваемый модуль имеет формальный ссылочный тип, то при предвыполнении соответствующей конкретизации проверяется совпадение любых ограничений на указанные объекты фактического и формального ссылочных типов. При несовпадении возбуждается исключение CONSTRAINT_ERROR.

Пример:

```
- - формальным типам настраиваемого пакета
generic
  type УЗЕЛ is private;
  type СВЯЗЬ is access УЗЕЛ;
package P is
  ...
end;
- - могут быть сопоставлены фактические типы
type МАШИНА;
type ИМЯ_МАШИНЫ is access МАШИНА;
type МАШИНА is
record
      ПРЕД, СЛЕД: ИМЯ_МАШИНЫ;
      НОМЕР: РЕГИСТРАЦИОННЫЙ_НОМЕР;
      ВЛАДЕЛЕЦ: ПЕРСОНА;
  end record;
```

```
- - в следующей конкретизации настройки
package R is new P (УЗЕЛ => МАШИНА, СВЯЗЬ => ИМЯ_МАШИНЫ);
```

Примечание. Если указанный тип сам является формальным, то, в соответствии с описанными выше правилами, в экземпляре его имя обозначает соответствующий фактический подтип (см. 12.3 г).

12.3.6. Правила сопоставления для формальных подпрограмм

Формальной подпрограмме сопоставляются фактическая подпрограмма, литерал перечисления или вход, если они имеют один и тот же профиль типов параметров и результата (см. 6.6), при этом виды формальных и фактических параметров в одинаковых позициях должны быть одинаковыми.

Если настраиваемый модуль имеет подпрограмму по умолчанию, заданную именем, то это имя должно обозначать подпрограмму, литерал перечисления или вход, сопоставленный формальной подпрограмме (в указанном выше смысле). Вычисление имени по умолчанию производится во время предвыполнения каждой конкретизации, в которой используется это умолчание, как определено в разд. 12.3.

Если настраиваемый модуль имеет подпрограмму по умолчанию, специфицированную как бокс, то соответствующий фактический параметр может быть опущен, если подпрограмма, литерал перечисления или вход, сопоставляемые формальной подпрограмме, имеют то же обозначение, что и формальная подпрограмма, и непосредственно на месте конкретизации должна быть видима единственная такая подпрограмма, или литерал перечисления, или вход.

Пример:

```
- - дана спецификация настраиваемой функции
generic
  type ЭЛЕМЕНТ is private;
  with function "*" (A, B: ЭЛЕМЕНТ) return ЭЛЕМЕНТ is < >;
function ВОЗВ_В_КВАДРАТ (X: ЭЛЕМЕНТ) return ЭЛЕМЕНТ;
- - и функция
function ПРОИЗВЕДЕНИЕ_МАТРИЦ (A, B: МАТРИЦА) return МАТРИЦА;
- - возможна следующая конкретизация
function КВАДРАТ os new ВОЗВ_В_КВАДРАТ (МАТРИЦА,
                                        ПРОИЗВЕДЕНИЕ_МАТРИЦ);
- - следующие конкретизации эквивалентны
function КВАДРАТ is new ВОЗВ_В_КВАДРАТ (ЭЛЕМЕНТ => INTEGER,
                    "*" => "*");
function КВАДРАТ is new ВОЗВ_В_КВАДРАТ (INTEGER, "*");
function КВАДРАТ is new ВОЗВ_В_КВАДРАТ (INTEGER);
```

Примечание. Правила сопоставления для формальных подпрограмм устанавливают требования, которые подобны требованиям, применяемым к описаниям переименования подпрограмм (см. 8.5). В частности, не требуется совпадения имен соответствующих параметров формальной и фактической подпрограмм; аналогично, для таких параметров не обязательно соответствие выражений по умолчанию.

Формальной подпрограмме сопоставляется атрибут типа, если этот атрибут – функция с сопоставимой спецификацией. Литерал перечисления данного типа сопоставляется с формальной функцией без параметров и результатом данного типа.

12.4. **Пример настраиваемого пакета**

В следующем примере использован настраиваемый пакет для одной из возможных организаций стеков. Размер каждого стека и тип его элементов являются параметрами настройки.

```
generic
  РАЗМЕР: POSITIVE;
  type ЭЛЕМЕНТ is private;
package СТЕК is
  procedure В_СТЕК (Е: in ЭЛЕМЕНТ);
  procedure ИЗ_СТЕКА (Е: out ЭЛЕМЕНТ);
  ПЕРЕПОЛНЕНИЕ, ПУСТОТА: exception;
end СТЕК;
package body СТЕК is
  type ТАБЛИЦА is array (POSITIVE range < >) of ЭЛЕМЕНТ;
  МЕСТО: ТАБЛИЦА (1. .РАЗМЕР);
  ИНДЕКС: NATURAL: = 0;
  procedure В_СТЕК (Е: in ЭЛЕМЕНТ) is
  begin
     if ИНДЕКС > = РАЗМЕР then
        raise ПЕРЕПОЛНЕНИЕ;
     end if;
     ИНДЕКС: = ИНДЕКС + 1;
     МЕСТО (ИНДЕКС) : = Е;
  end В_СТЕК;
  procedure ИЗ_СТЕКА (Е: out ЭЛЕМЕНТ) is
  begin
     if ИНДЕКС = 0 then
        raise ПУСТОТА;
     end if;
        Е: = МЕСТО (ИНДЕКС);
        ИНДЕКС: = ИНДЕКС – 1;
     end ИЗ_СТЕКА;
end СТЕК;
```

Экземпляры настраиваемого пакета могут быть получены так:

```
package СТЕК_ЦЕЛ is new СТЕК (РАЗМЕР => 200, ЭЛЕМЕНТ => INTEGER);
package СТЕК_ЛОГ is new СТЕК (100, BOOLEAN):
```

После этого могут быть вызваны процедуры конкретизированных пакетов:

```
СТЕК_ЦЕЛ.В_СТЕК (К);
СТЕК_ЛОГ. В_СТЕК (TRUE);
```

Возможна другая организация стека (тело пакета опущено):

```
generic
  type ЭЛЕМЕНТ is private;
package РАБОТА_СО_СТЕКОМ is
  type СТЕК (РАЗМЕР: POSITIVE) is limited private;
  procedure В_СТЕК (С: in out СТЕК; Е: in ЭЛЕМЕНТ);
  procedure ИЗ_СТЕКА (С: in out СТЕК; Е: out ЭЛЕМЕНТ);
  ПЕРЕПОЛНЕНИЕ, ПУСТОТА: exception;
private
  type ТАБЛИЦА is array (POSITIVE range < >) of ЭЛЕМЕНТ;
  type СТЕК (РАЗМЕР: POSITIVE) is
     record
```

```
        МЕСТО: ТАБЛИЦА (1..РАЗМЕР);
        ИНДЕКС: NATURAL: = 0;
    end record;
end;
```

При использовании такого пакета сначала должна быть осуществлена конкретизация, после чего можно описать стеки с элементами соответствующего типа:

```
declare
  package СТЕК_ВЕЩЕСТВ is new РАБОТА_СО_СТЕКОМ (ВЕЩЕСТВ);
  use СТЕК_ВЕЩЕСТВ;
  С: СТЕК (100);
begin
  ...
  В_СТЕК (С, 2.54);
end;
```

13. СПЕЦИФИКАТОРЫ ПРЕДСТАВЛЕНИЯ И ОСОБЕННОСТИ, ЗАВИСЯЩИЕ ОТ РЕАЛИЗАЦИИ

В этой главе описываются спецификаторы представления, зависящие от реализации особенности, а также некоторые возможности, используемые в системном программировании.

13.1. Спецификаторы представления

Спецификаторы представления задают способ представления типов в объектной машине для более эффективного представления или для интерфейса с внеязыковой сферой (например, с периферийным оборудованием).

спецификатор_представления : : =
  спецификатор_представления_типа | спецификатор_адреса
спецификатор_представления_типа : : = спецификатор_длины
  | спецификатор_представления_перечисления
  | спецификатор_представления_записи

Спецификатор представления типа применяется либо к типу, либо к *первому именованному подтипу* (т. е. подтипу, заданному описанием типа, базовый тип которого является анонимным). Такой спецификатор представления применяется ко всем объектам данного типа или данного первого именованного подтипа. Для конкретного типа допустимо не более одного спецификатора представления перечисления или записи: спецификатор представления перечисления допустим только для перечислимого типа; спецификатор представления записи – только для именуемого типа. (С другой стороны, для конкретного типа может быть задано более одного спецификатора длины; более того, могут быть одновременно заданы спецификатор длины и спецификатор представления записи или перечисления.) Спецификатор длины – это единственный из спецификаторов представления, допустимый для производного от родительского типа, имеющего (определенные пользователем) наследуемые подпрограммы.

Спецификатор адреса применяется либо к объекту, либо к подпрограмме, пакету или задачному модулю, либо к входу. Для любого из этих понятий допустимо не более одного спецификатора адреса.

Спецификатор представления и описание понятия, к которому применяется спецификатор, должны оба находиться непосредственно в одном и том же разделе описаний, спецификации пакета или спецификации задачи; описание должно помещаться до спецификатора. В отсутствие спецификатора представления для данного описания реализация определяет представление по умолчанию. Место нахождения такого подразумеваемого определения представления по умолчанию – не позже конца непосредственно объемлющего раздела описаний, спецификаций пакета или задачи. Для описания из раздела описаний место нахождения подразумеваемого определения по умолчанию – до любого вложенного тела.

В случае типа некоторые вхождения его имени неявно предполагают, что представление типа уже должно быть определено. Следовательно, такие вхождения требуют определения по умолчанию любого способа представления, еще не определенного предшествующим спецификатором представления типа. Аналогичные вхождения имени подтипа этого типа или имени любого типа или подтипа с подкомпонентами данного типа также требуют определения по умолчанию. Требующее вхождение – это любое вхождение, отличное от вхождения в описание типа или подтипа, спецификацию подпрограммы, описание входа, описание субконстанты, прагму или спецификатор представления для самого типа. В любом случае вхождение в выражение является всегда требующим.

Спецификатор представления для данного понятия не должен помещаться после вхождения имени понятия, если вхождение требует определения представления этого понятия по умолчанию.

Аналогичные ограничения существуют для спецификатора адреса. Любое вхождение имени объекта (после описания объекта) требует определения представления. Для подпрограммы, пакета, задачного модуля или входа любое вхождение атрибута представления таких понятий является требуемым вхождением.

Результат предвыполнения спецификатора представления – определение соответствующих аспектов представления.

Интерпретация некоторых выражений, помещенных в спецификаторах представления, зависит от реализации, например, выражений, задающих адреса. Реализация может ограничивать использование спецификаторов представления лишь теми, которые можно просто обрабатывать на имеющемся оборудовании. Для учитываемых реализацией спецификаторов представления компилятор должен гарантировать независимость конечного результата работы программы от наличия или отсутствия таких спецификаторов представления, исключая спецификатор адреса и те разделы программы, где используются атрибуты представления. Если программа содержит спецификатор представления, который не учитывается реализацией, она неправильна. Для каждой реализации в обязательном приложении 4 даются правила составления руководства по языку, в котором должны быть описаны допустимые ею спецификаторы представления и соглашения для выражений, зависящих от реализации.

Если спецификатор представления используется для того, чтобы предписывать некоторые характеристики отображения понятия в объектной машине, то могут быть использованы прагмы, чтобы реализация обеспечила выбор такого отображения. Прагма PACK указывает, что минимизация размера памяти является главным критерием при выборе представления именуемого или индексируемого типов. Ее форма такова:

pragma PACK (простое_имя_*типа*);

Упаковка означает, что промежутки между областями памяти, выделенные под последовательные компоненты, следует минимизировать. Упаковка не влияет на отображение в памяти каждого компонента. На отображение компонентов можно повлиять прагмой (или спецификатором представления) для компонента или типа компонента. Место прагмы PACK в программе и ограничение на именуемый тип подчинены тем же правилам, что и для спецификатора представления; в частности, прагма должна помещаться до любого использования атрибута представления упакованного понятия.

Прагма PACK – единственная определенная в языке прагма, связанная с представлением. Реализация может вводить дополнительные прагмы; они должны быть перечислены в руководстве по реализации в соответствии с обязательным приложением 4. (В отличие от спецификаторов представления, прагма, которая не принята в реализации, игнорируется.)

Примечание. Для формального типа настройки не допустим спецификатор представления.

13.2. **Спецификаторы длины**

Спецификатор длины задает объем памяти, выделяемой для объектов указанного типа.

спецификатор_длины : : = **for** атрибут **use** простое_выражение;

Выражение должно быть некоторого числового типа; оно вычисляется при предвыполнении спецификатора длины (кроме случая, когда оно является статическим). Префикс атрибута должен обозначать либо тип, либо первый именованный подтип. Далее этот префикс обозначается буквой T. В спецификаторе длины в качестве обозначений атрибутов допустимы только SIZE, STORAGE_SIZE и SMALL. Результат применения спецификатора длины зависит от обозначения атрибута:

а) Спецификация размера: T'SIZE

Выражение должно быть статическим выражением некоторого целого типа. Его значение определяет максимальное число битов, используемых для размещения в памяти объектов типа (или первого именованного подтипа) T. Спецификация размера должна задавать объем памяти, достаточный для размещения любого допустимого значения таких объектов. Спецификация размера для составного типа может повлиять на размер промежутков между областями памяти, отводимыми под последовательные компоненты. С другой стороны, нет необходимости влиять на размер области памяти, отводимой для каждого компонента.

Спецификация размера допустима, только если ограничения на T и его компоненты (если они есть) являются статическими. Для неограниченного

индексируемого типа статическими должны быть также подтипы индексов.

б) Спецификация размера набора: T'STORAGE_SIZE

Префикс T должен обозначать ссылочный тип. Выражение должно быть некоторого целого типа (но не обязательно статическим); его значение определяет число квантов памяти, выделяемых для набора, т. е. объем памяти, необходимый для размещения всех объектов, указанных значениями ссылочного типа и значениями других типов, непосредственно или косвенно производных от ссылочного типа. Эта форма спецификатора длины недопустима для типа, производного от ссылочного типа.

в) Спецификация объема памяти для активизации задачи:

T'STORAGE_SIZE

Префикс T должен обозначать задачный тип. Выражение должно быть некоторого целого типа (но не обязательно статическим); его значение определяет число квантов памяти, выделяемых для активизации (но не для кода) задачи данного типа.

г) Спецификация *дискрета* для фиксированных типов: T'SMALL

Префикс T должен обозначать первый именованный подтип фиксированного типа. Выражение должно быть статическим некоторого вещественного типа; его значение должно быть не больше чем дельта этого первого поименованного подтипа. Результат применения спецификатора длины – использование этого *дискрета* значения для представления значений фиксированного базового типа. (Спецификатор длины, таким образом, также влияет на выделяемый для объектов этого типа объем памяти.)

Примечание. Спецификация размера допустима для ссылочного, задачного или фиксированного типов независимо от того, задана или нет для такого типа какая-либо другая форма спецификатора длины.

От реализации зависит, что понимается под резервированием части памяти для набора или активизации задачи. Поэтому управление с помощью спецификаторов длины зависит от соглашений в реализации. Например, в языке не определено, включается ли в память, выделяемую для активизации задачи, память, необходимая для размещения набора, сопоставленного ссылочному типу, описанному в теле задачи. Метод распределения памяти под объекты, обозначенные значениями ссылочного типа, также не определяется. Например, место может выделяться в стеке, можно использовать схему настраиваемого динамического распределения памяти или может быть использована фиксированная память.

Размещенные объекты набора не обязаны занимать одинаковые размеры памяти, если указанный тип – это неограниченный индексируемый тип или неограниченный тип с дискриминантами. Заметим также, что сам генератор может требовать некоторый объем памяти для размещения внутренних таблиц и связей. Следовательно, спецификатор длины для набора ссылочного типа не всегда позволяет точно управлять максимальным числом генерируемых объектов.

Примеры:

```
- - предполагаемые описания:
type СРЕДА is range 0. .65000;
type КОРОТКИЙ is delta 0.01 range – 100.0. .100.0;
type ГРАДУС is delta 0.1 range – 360.0. .360.0;
БАЙТ: constant: = 8;
```

```
СТРАНИЦА: constant: = 2000;
- - спецификаторы длины:
for ЦВЕТ'SIZE use 1 * БАЙТ; - - см. 3.5.1
for СРЕДА'SIZE use 2 * БАЙТ;
for КОРОТКИЙ'SIZE use 15;
for ИМЯ_МАШИНЫ' STORAGE_SIZE use - - примерно 2000 машин
    2000 * ((МАШИНА'SIZE/SYSTEM. STORAGE_UNIT) + 1);
for КЛАВИАТУРА'STORAGE_SIZE use 1 * СТРАНИЦА;
for ГРАДУС'SMALL use 360.0/2 ** (SYSTEM. STORAGE_UNIT –1);
```

Примечание к примерам. В спецификаторе длины для КОРОТКИЙ минимально необходимы пятнадцать разрядов, так как определение типа требует КОРОТКИЙ'SMALL = 2.0 ** (–7) и КОРОТКИЙ'MANTISSA = 14.

Спецификатор длины для ГРАДУС вводит модельные числа, которые занимают в точности весь диапазон типа.

13.3. Спецификаторы представления перечисления

Спецификатор представления перечисления задает внутренние коды для литералов перечислимого типа, указанного в спецификаторе.

спецификатор_представления_перечисления : : =

for простое_имя_*типа* **use** агрегат;

Используемый в спецификаторе агрегат записывается как одномерный агрегат, в котором подтип индекса – перечислимый тип, а тип компонента – *универсальный_целый* тип.

Для всех литералов перечислимого типа должны быть заданы различные целые коды, и все выборы и значения всех компонентов в агрегате должны быть статическими. Целые коды, заданные для перечислимого типа, должны удовлетворять предопределенному отношению упорядоченности типа.

Пример:

```
type МАШ_КОДЫ is (СЛОЖ, ВЫЧ, УМН, ЗАГР, ВЫГР, ВЫГР_ЗН);
for МАШ_КОДЫ use
  (СЛОЖ => 1, ВЫЧ => 2, УМН => 3, ЗАГР => 8, ВЫГР => 24, ВЫГР_ЗН => 33);
```

Примечание. Атрибуты SUCC, PRED и POS определены даже для перечислимых типов с разрывным представлением; их определение соответствует (логическому) описанию типа и на них не влияет спецификатор представления перечисления. В примере из-за того, что значения даны с пропуском, эти функции реализуются менее эффективно, чем это могло быть в отсутствие спецификатора представления. Это же справедливо и при использовании таких типов для индексации.

13.4. Спецификаторы представления записей

Спецификатор представления записи задает представление записи в памяти, т. е. порядок, позицию и размер компонентов записи (включая дискриминанты, если они есть).

спецификатор_представления_записи : : =

```
  for простое_имя_типа use
  record [спецификатор_выравнивания]
     {спецификатор_компонента}
  end record;
```

спецификатор_выравнивания : : =

at mod *статическое*_простое_выражение;
спецификатор_компонента : : =
имя_*компонента* **at** *статическое*_простое_выражение
range *статический*_диапазон;

Простое выражение, заданное в спецификаторе выравнивания после зарезервированных слов **at mod** или в спецификаторе компонента после зарезервированного слова **at**, должно быть статическим некоторого целого типа. Если в спецификаторе компонента границы диапазона спецификатора компонента определяются простыми выражениями, то каждая граница должна быть определена как статическое выражение любого целого типа; необязательно, чтобы обе границы были одного и того же целого типа.

Спецификатор выравнивания требует, чтобы каждая запись данного типа была размещена, начиная с адреса памяти, кратного значению данного выражения (т. е. адрес по модулю выражения должен быть равен нулю). Реализация может наложить ограничения на допускаемые выравнивания.

Спецификатор компонента специфицирует для компонента *место в памяти* относительно начала записи. Значение целого типа, определенное статическим выражением в спецификаторе компонента, — это относительный адрес, выраженный в квантах памяти. Диапазон определяет позиции разрядов места памяти относительно этого кванта памяти. Первый квант памяти для записи имеет нулевой номер. Первый разряд кванта памяти тоже имеет нулевой номер. Порядок разрядов в кванте памяти машинно-зависим, а их нумерация может переходить на соседние кванты. (Для конкретной машины размер кванта памяти в разрядах задан с помощью зависящего от конфигурации именованного числа SYSTEM.STORAGE_ UNIT.) Допускается размещение одного компонента записи в соседних квантах памяти, это размещение определяется реализацией.

Для каждого компонента именуемого типа, включая каждый дискриминант, допустимо не более одного спецификатора компонента. Спецификаторы компонентов могут быть даны для нескольких, всех или ни для одного из компонентов. Если для компонента не задан спецификатор компонента, то выбор места в памяти для компонента определяется компилятором. Если спецификаторы компонентов даны для всех компонентов, то спецификатор представления записи полностью задает представление именуемого типа, и компилятор должен в точности следовать спецификатору.

Места в памяти для компонентов в пределах одного варианта не должны перекрываться, но допускается перекрытие для различных вариантов. Каждый спецификатор компонента должен допускать достаточный размер памяти для размещения каждого допустимого значения компонента. Спецификатор компонента допустим только для такого компонента, для которого любое ограничение, наложенное на него или на его подкомпоненты, является статическим.

Реализация может генерировать имена, обозначающие зависящие от реализации компоненты (например, компонент, содержащий смещение другого компонента). Такие имена могут быть использованы в спецификаторах

представления записей (эти имена могут не быть простыми именами, например, они могут быть зависящими от реализации атрибутами).

Пример:

```
СЛОВО: constant: = 4; -- квант памяти – это байт; в слове 4 байта
type СОСТОЯНИЕ is (A, M, W, P);
type ВИД is (ФИКС, ДЕСЯТ, ПОКАЗ, ЗНАК);
type МАСКА_БАЙТА is array (0..7) of BOOLEAN;
type МАСКА_СОСТОЯНИЯ is array (СОСТОЯНИЕ) of BOOLEAN;
type МАСКА_ВИДА is array (ВИД) of BOOLEAN;
type СЛОВО_СОСТОЯНИЯ_ПРОГРАММЫ is
 record
    МАСКА_СИСТЕМЫ              : МАСКА_БАЙТА;
    КЛЮЧ_ЗАЩИТЫ                : INTEGER   range 0..3;
    СОСТОЯНИЕ_МАШИНЫ           : МАСКА_СОСТОЯНИЯ;
    ПРИЧИНА_ПРЕРЫВАНИЯ         : КОД_ПРЕРЫВАНИЯ;
    ПРИЗНАК_1                  : INTEGER   range 0..3
    ПРИЗНАК_2                  : INTEGER   range 0..3
    МАСКА_ПРОГРАММЫ            : МАСКА_ВИДА;
    ТЕК_АДРЕС                  : ADDRESS;
 end record;
for СЛОВО_СОСТОЯНИЯ_ПРОГРАММЫ use
 record at mod 8;
    МАСКА_СИСТЕМЫ at 0 * СЛОВО range  0..7; -- 8 и 9 разряды не используются
    КЛЮЧ_ЗАЩИТЫ at 0 * СЛОВО range 10..11;
    СОСТОЯНИЕ_МАШИНЫ at 0 * СЛОВО range 12..15;
    ПРИЧИНА_ПРЕРЫВАНИЯ at 0 * СЛОВО range 16..31; -- второе слово
    ПРИЗНАК_1 at  1 * СЛОВО range 0..1;
    ПРИЗНАК_2 at 1 * СЛОВО range 2..3;
    МАСКА_ПРОГРАММЫ at  1 * СЛОВО range 4..7;
    ТЕК  АДРЕС at 1 * СЛОВО range 8..31;
 end record;
for СЛОВО_СОСТОЯНИЯ_ПРОГРАММЫ'SIZE use
    8 * SYSTEM.STORAGE_UNIT;
```

Примечание к примеру. Спецификатор представления записи определяет размещение полей записи. Спецификатор длины гарантирует, что при этом будет использовано точно восемь квантов памяти.

13.5. **Спецификаторы адреса**

Спецификатор адреса задает для понятия адрес в памяти.

спецификатор_адреса : : = **for** простое_имя **use at** простое_выражение;

Выражение после зарезервированного слова **at** должно быть типа ADDRESS, определенного в пакете SYSTEM (см. разд. 13.7). Этот пакет должен быть упомянут в спецификаторе совместности компилируемого модуля, в котором помещается спецификатор адреса. Условия, определяющие интерпретацию значения типа ADDRESS как адреса, уровня прерывания или чего-либо другого, зависят от реализации. Ниже перечислены допустимые толкования простого имени и смысл соответствующего адреса:

а) Имя объекта: требуемый адрес – адрес объекта (переменной или константы).

б) Имя подпрограммы, пакета или задачного модуля: требуемый адрес – адрес машинного кода, связанного с телом программного модуля.

в) Имя одиночного входа: адрес задает аппаратное прерывание, связанное с этим входом.

Если простое имя – это имя одиночной задачи, то спецификатор адреса относится к задачному модулю, а не к задачному объекту. Во всех случаях спецификатор адреса является правильным только тогда, когда точно одно описание с этим идентификатором помещено перед ним непосредственно в том же разделе описаний, спецификации пакета или спецификации задачи. В качестве простого имени не допустимо имя, введенное описанием переименования.

Спецификаторы адреса не могут быть использованы для задания перекрытия объектов или перекрытия программных модулей. Данное прерывание может быть связано не более чем с одним входом. Любая программа, использующая такой эффект спецификатора адреса, ошибочна.

Пример:

```
for УПРАВЛЕНИЕ use at 16 # 0020 #;
-- предполагается, что SYSTEM. ADDRESS - это некоторый целый тип
```

Примечание. Из приведенных правил следует, что если в данной точке программы видимы две совмещенные подпрограммы, то спецификатор адреса для каждой из этих подпрограмм является неправильным в данной точке. Аналогично, если в спецификации задачи описаны совмещенные друг с другом входы, то они не могут быть входами обработки прерываний. Для библиотечного модуля синтаксис не допускает спецификатора адреса. Реализацией могут быть определены прагмы для обеспечения оверлейных перекрытий программ.

13.5.1. Прерывания

Спецификатор адреса для входа сопоставляет вход с некоторым устройством, которое может вызвать прерывание; такой вход в этом разделе называется *входом по прерыванию.* Если прерывание сопровождается управляющей информацией, то она передается соответствующему входу по прерыванию как один или несколько параметров входа вида **in**; допустимы параметры только такого вида.

Прерывание действует как вызов входа некоторой связанной с оборудованием задачи, приоритет которой выше приоритета главной программы и любых определенных пользователем задач (т. е. любой задачи, тип которой описан с помощью задачного модуля в программе). Вызов входа может быть обычным, временным или условным вызовом входа в зависимости от вида прерывания и от реализации.

Если оператор отбора содержит как альтернативу завершения, так и альтернативу принятия входа по прерыванию, то реализация может наложить некоторые дополнительные требования на отбор альтернативы завершения в дополнение к требованиям, сформулированным в разд. 9.4.

Пример:

```
task ОБРАБОТКА_ПРЕРЫВАНИЯ is
  entry ГОТОВЫЙ;
  for ГОТОВЫЙ use at 16 # 40 #;
  -- предполагается, что SYSTEM.ADDRESS – это некоторый целый тип
end ОБРАБОТКА_ПРЕРЫВАНИЯ;
```

Примечание. Вызовы входа по прерыванию имеют только описанную выше семантику; они могут быть реализованы с помощью аппаратуры, непосредственно выполняющей соответствующие операторы принятия.

Выстраиваемые в очередь прерывания соответствуют обычным вызовам входа. Прерывания, которые теряются, если немедленно не обрабатываются, соответствуют условным вызовам входов. Из правил приоритетов следует, что оператор принятия, выполняемый в ответ на прерывание, имеет более высокий приоритет, чем определенные пользователями обычные задачи, и может быть выполнен без задачи, выполняющей планировку.

Один из возможных результатов указания спецификатора адреса для входа по прерыванию является спецификация (непосредственно или косвенно) приоритета прерывания. Допустимы прямые вызовы входов по прерыванию.

13.6. **Изменение представления**

Для данного типа и для данного аспекта его представления допустимо не более одного спецификатора представления. Поэтому, если желательно другое представление, то необходимо описать второй тип, производный от первого, и для него специфицировать другое представление.

Пример:

```
- - УПАКОВАННЫЙ_ДЕСКРИПТОР и ДЕСКРИПТОР – это два различных
- - типа с одинаковыми характеристиками, но различным представлением.
type ДЕСКРИПТОР is
  record
  - - компоненты дескриптора
  end record;
type УПАКОВАННЫЙ_ДЕСКРИПТОР is new ДЕСКРИПТОР;
for УПАКОВАННЫЙ_ДЕСКРИПТОР use
  record
  - - спецификаторы компонентов для всех или некоторых компонентов
  end record;
```

Изменение представления может быть теперь достигнуто присваиванием с явным преобразованием типа:

```
Д: ДЕСКРИПТОР;
У: УПАКОВАННЫЙ_ДЕСКРИПТОР;
У: = УПАКОВАННЫЙ_ДЕСКРИПТОР (Д); - - упаковка Д
Д: = ДЕСКРИПТОР (У); - - распаковка У
```

13.7. **Системный пакет**

Для каждой реализации имеется предопределенный библиотечный пакет SYSTEM, который включает определения некоторых характеристик, зависящих от конфигурации. Спецификация пакета зависит от реализации и должна быть приведена в руководстве по реализации в соответствии с обязательным приложением 4. Видимый раздел этого пакета должен содержать по крайней мере следующие описания:

```
package SYSTEM is
  type ADDRESS is определен_реализацией;
  type NAME is определен_реализацией_перечислимого_типа;
  SYSTEM_NAME: constant NAME: = определен_реализацией;
  STORAGE_UNIT: constant: = определен_реализацией;
  MEMORY_SIZE: constant: = определен_реализацией;
  - - зависящие от системы именованные числа:
  MIN_INT      : constant: = определен_реализацией;
  MAX_INT      : constant: = определен_реализацией;
  MAX_DIGITS   : constant: = определен_реализацией;
```

```
   MAX_MANTISSA : constant: = определен_реализацией;
   FINE_DELTA   : constant: = определен_реализацией;
   TICK         : constant: = определен_реализацией;
   - - другие зависящие от системы описания:
   subtype PRIORITY is INTEGER range определен_реализацией;
   . . .
end SYSTEM;
```

Тип ADDRESS – это тип адресов, задаваемых спецификаторами адреса; к этому же типу принадлежат значения, вырабатываемые атрибутом ADDRESS. Значения перечислимого типа NAME – это имена альтернативных машинных конфигураций, обрабатываемых реализацией; одно из них – константа SYSTEM_NAME. Именованное число STORAGE_UNIT равно числу разрядов в кванте памяти, а именованное число MEMORY_SIZE – числу квантов памяти, доступных в конфигурации; эти именованные числа имеют *универсальный_целый* тип.

Альтернативная форма пакета SYSTEM с другими значениями SYSTEM_NAME, STORAGE_UNIT и MEMORY_SIZE может быть получена использованием соответствующих прагм. Эти прагмы допустимы только в начале компиляции до первого компилируемого модуля (если он есть) компиляции.

pragma SYSTEM_ NAME (литерал_перечисления);

В результате выполнения этой прагмы заданный идентификатором литерал перечисления будет использован для определения константы SYSTEM_NAME. Эта прагма допустима, только если этот идентификатор соответствует одному из литералов типа NAME.

pragma STORAGE_UNIT (числовой_литерал);

В результате выполнения этой прагмы заданное числовым литералом значение будет использовано для определения именованного числа STORAGE_UNIT.

pragma MEMORY_SIZE (числовой_литерал);

В результате выполнения этой прагмы заданное числовым литералом значение будет использовано для определения именованного числа MEMORY_SIZE.

Компиляция любой из этих прагм вызовет неявную перекомпиляцию пакета SYSTEM. Следовательно, любой компилируемый модуль, в спецификаторе контекста которого упоминается пакет SYSTEM, становится устаревшим. Реализация может ввести дополнительные ограничения на использование этих прагм. Например, реализация может допустить их только в начале первой компиляции, когда создается новая программная библиотека.

Примечание. Согласно правилам видимости описание из пакета SYSTEM видимо в компилируемом модуле только в том случае, если этот пакет упомянут в спецификаторе совместности, примененном (непосредственно или косвенно) к данному компилируемому модулю.

13.7.1. Зависящие от системы именованные числа

Перечисленные ниже именованные числа описаны в пакете SYSTEM. Числа FINE_DELTA и TICK принадлежат *универсальному_вещественному* типу, остальные – *универсальному_целому* типу.

MIN_INT Наименьшее (наибольшее по модулю отрицательное) значение из всех определенных целых типов.

MAX_INT Наибольшее (положительное) значение из всех предопределенных целых типов.

MAX_DIGITS Наибольшее допустимое значение числа значащих десятичных цифр в ограничении для плавающего типа.

MAX_MANTISSA Наибольшее возможное число двоичных цифр в мантиссе модельных чисел фиксированного подтипа.

FINE_DELTA Наименьшая дельта, допустимая в ограничении для фиксированного типа, который имеет ограничение диапазона – 1.0. .1.0.

TICK Базовый период времени, выраженный в секундах.

13.7.2. Атрибуты представления

Значения некоторых зависящих от реализации характеристик могут быть получены с помощью соответствующих *атрибутов представления*. Эти атрибуты описаны ниже.

Для любого объекта, программного модуля, метки или входа X:

X'ADDRESS Вырабатывает адрес первого кванта памяти, отведенной под X. Для подпрограммы, пакета, задачного модуля или метки это значение ссылается на машинный код, связанный с соответствующим телом или оператором. Для входа, для которого задан спецификатор адреса, это значение ссылается на соответствующее аппаратное прерывание. Значение этого атрибута принадлежит типу ADDRESS, определенному в пакете SYSTEM.

Для любого типа или подтипа X или для любого объекта X:

X'SIZE Примененный к объекту вырабатывает число битов, отводимых в памяти для размещения объекта. Примененный к типу или подтипу вырабатывает минимальное число битов, необходимое реализации для размещения любого возможного объекта этого типа или подтипа. Значение этого атрибута имеет тип *универсальный_целый*.

Если префиксом атрибута является функция, то атрибут понимается как атрибут функции (а не результата вызова функции). Если тип префикса – *ссылочный тип, то атрибут понимается как атрибут префикса (а не* указанного объекта: атрибуты этого объекта могут быть записаны с префиксом, оканчивающимся зарезервированным словом **all**).

Для любого компонента K записи З:

З.K'POSITION Вырабатывает величину смещения первого кванта памяти, занятого полем K, относительно начала первого кванта памяти, занятого записью З. Величина смещения измеряется числом квантов памяти. Значение этого атрибута принадлежит *универсальному_целому* типу.

З.K'FIRST_BIT Вырабатывает величину смещения первого бита, занятого полем K, относительно начала первого кванта памяти, занятого K. Величина смещения измеряется числом битов. Значение этого атрибута имеет *универсальный_целый* тип.

З.K'LAST_BIT Вырабатывает величину смещения последнего бита, занятого полем K, относительно начала первого кванта памяти, занятого K. Величина смещения измеряется числом битов. Значение этого атрибута имеет *универсальный_целый* тип.

Для любого ссылочного типа или подтипа T:

T'STORAGE_SIZE Вырабатывает общее число квантов памяти, выделенных для набора, связанного с базовым типом T. Значение атрибута имеет *универсальный_целый* тип.

Для любого задачного типа или объекта задачного типа T:

T'STORAGE_SIZE Вырабатывает число квантов памяти, выделенных для каждой активизации задачи типа T или для активизации объекта T задачного типа. Значение этого атрибута имеет *универсальный_целый* тип.

Примечание. Для объекта X задачного типа атрибут X'SIZE вырабатывает число разрядов, используемых для размещения объекта X; атрибут X'STORAGE_SIZE вырабатывает число квантов памяти, выделенных для активизации задачи, указанной X. Для формального параметра X в случае передачи параметра копированием X'ADDRESS вырабатывает адрес локальной копии; в случае передачи параметра ссылкой X'ADDRESS вырабатывает адрес фактического параметра.

13.7.3. Атрибуты представления вещественных типов

Для каждого вещественного типа или подтипа T определены нижеследующие машинно-зависимые атрибуты, не связанные с модельными числами. Использующие эти атрибуты программы могут получить некоторую дополнительную информацию о характеристиках числового типа (см. разд. 4.5.7 о правилах определения точности операций с вещественными операндами). Для обеспечения переносимости программ должна быть обеспечена известная осторожность в использовании таких машинно-зависимых атрибутов.

Атрибуты, применимые к плавающим и фиксированным типам:

T'MACHINE_ROUNDS Вырабатывает значение TRUE, если каждая предопределенная арифметическая операция над значениями базового типа T либо возвращает точный результат, либо осуществляет округление. В противном случае вырабатывает значение FALSE. Значение этого атрибута имеет предопределенный тип BOOLEAN.

T'MACHINE_OVERFLOWS Вырабатывает значение TRUE, если каждая предопределенная операция над значениями базового типа T либо возвращает точный результат, либо возбуждает исключение NUMERIC_ERROR при переполнении (см. разд. 4.5.7); в противном случае вырабатывает значение FALSE. Значение этого атрибута имеет предопределенный тип BOOLEAN.

Следующие атрибуты дают характеристики машинного представления значений плавающего типа в терминах канонической формы, определенной в разд. 3.5.7:

T'MACHINE_RADIX Вырабатывает значение *основания* системы счисления, используемого в машинном представлении базового типа T. Значение этого атрибута имеет *универсальный_целый* тип.

T'MACHINE_MANTISSA Вырабатывает число цифр в *мантиссе* машинного представления базового типа T. (Цифра – это расширенная цифра из диапазона от 0 до T'MACHINE_RADIX–1.) Значение этого атрибута имеет *универсальный_целый* тип.

T'MACHINE_EMAX Вырабатывает наибольшее значение *порядка* в машинном представлении базового типа T. Значение этого атрибута имеет *универсальный_целый* тип.

T'MACHINE_EMIN Вырабатывает наименьшее (наибольшее по модулю отрицательное) значение *порядка* в машинном представлении базового типа T. Значение этого атрибута имеет *универсальный_целый* тип.

Примечание. В большинстве машин наибольшее представимое в машине число типа *T* равно

(T'MACHINE_RADIX) ** (T' MACHINE_EMAX),

а наименьшее положительное представимое число в машине равно

(T'MACHINE_RADIX) ** (T'MACHINE_EMIN–1).

13.8. **Вставки машинных кодов**

Машинные коды могут быть включены в программу с помощью вызова процедуры, последовательность операторов которой состоит из операторов кода.

оператор_кода : : = обозначение_типа агрегат_*записи;*

Оператор кода допустим только в последовательности операторов тела процедуры. Если в теле процедуры содержатся операторы кода, то в нем недопустимы никакие формы операторов, кроме операторов кода (помеченных или нет); из описаний допустимы только спецификаторы использования; недопустимы обработчики исключения (комментарии и прагмы допустимы как обычно).

Каждая машинная команда записывается как агрегат именуемого типа, агрегат определяет эту команду. Базовый тип обозначения типа в операторе кода должен быть описан в предопределенном пакете MACHINE_CODE; этот пакет должен упоминаться в спецификаторе контекста, применяемом к компилируемому модулю, в который входит оператор кода. Реализация не обязана обеспечивать такой пакет.

В реализации допустимо наложение дополнительных ограничений на допустимые в операторах кода агрегаты записи. Например, можно требовать, чтобы выражения в агрегатах были статическими.

Реализация может определить машинно-зависимые прагмы, указывающие соглашения об использовании регистров и вызовов. Такие соглашения и прагмы должны быть описаны в руководстве по реализации в соответствии с обязательным приложением 4.

Пример:

```
M: МАСКА;
procedure УСТАНОВИТЬ_МАСКУ;
pragma INLINE (УСТАНОВИТЬ_МАСКУ);
procedure УСТАНОВИТЬ_МАСКУ is
  use MACHINE_CODE;
begin
  ФОРМАТ_СИ'(КОД => SSM, Б => М'БАЗОВЫЙ_РЕГ, СМ => М'СМЕЩ);
  - - М'БАЗОВЫЙ_РЕГ и М'СМЕЩ – это заданные реализацией предопределенные
  - - атрибуты
end;
```

13.9. **Связь с другими языками**

Из программы, написанной на языке Ада, может быть вызвана подпрограмма, написанная на другом языке; все связи с этими подпрограммами обеспечиваются через параметры и результаты функций. Для каждой такой подпрограммы должна быть задана следующая прагма:

pragma INTERFACE (*имя_языка*, *имя_подпрограммы*);

Допустимо использование совмещенных имен подпрограмм. Эта прагма допустима на месте элемента описания и должна применяться к подпрограмме, описанной ранее в этом же разделе описаний или спецификации пакета. Прагма также допустима и для библиотечного модуля; в этом случае прагма должна помещаться после описания подпрограммы, но до любого следующего компилируемого модуля. Прагма задает другой язык (и тем самым соглашения о вызовах) и сообщает компилятору, что для такой подпрограммы будет задан объектный модуль. Для таких подпрограмм недопустимо задание тела (даже в форме следа тела), так как его команды написаны на другом языке.

Эту возможность не обязательно обеспечивают все реализации. Реализация может наложить ограничения на допускаемые формы и места параметров и вызовов.

Пример:

```
package БИБЛ_ФОРТ is
  function SQRT (X: FLOAT) return FLOAT;
  function EXP (X: FLOAT) return FLOAT;
private
  pragma INTERFACE (ФОРТРАН, SQRT);
  pragma INTERFACE (ФОРТРАН, EXP);
end БИБЛ_ФОРТ;
```

Примечание. Соглашения, использованные в других языковых процессорах, которые вызывают Ада-программы, не являются частью определения языка Ада. Эти соглашения должны быть определены в описании других языковых процессоров.

Прагма INTERFACE не определена для настраиваемых подпрограмм.

13.10. **Неконтролируемое программирование**

Для неконтролируемого освобождения памяти и для неконтролируемого преобразования типов используются предопределенные настраиваемые библиотечные подпрограммы: UNCHECKED_DEALLOCATION и UNCHECKED_CONVERSION.

```
generic
  type OBJECT is limited private;
  type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION (X: in out NAME);
generic
  type SOURCE is limited private;
  type TARGET is limited private;
function UNCHECKED_CONVERSION (S: SOURCE) return TARGET;
```

13.10.1. *Неконтролируемое освобождение памяти*

В результате вызова процедуры, полученной конкретизацией настраиваемой процедуры UNCHECKED_DEALLOCATION, производится неконт-

ролируемое освобождение памяти, занимаемой объектом, указанным значением ссылочного типа. Например:

```
procedure СВОБОДНО is new UNCHECKED_DEALLOCATION
        (имя_типа_объекта, имя_ссылочного_типа);
```

Такая процедура СВОБОДНО дает следующий результат:

а) после выполнения СВОБОДНО (X) значением X является **null**;

б) если X уже равно **null**, то СВОБОДНО (X) не имеет другого результата;

в) если X не равно **null**, то СВОБОДНО (X) обозначает, что указанный значением X объект не требуется, и поэтому занимаемая им память может использоваться для других целей.

Если X и Y указывают на один и тот же объект, то после вызова СВОБОДНО (X) доступ к этому объекту (или попытка доступа к нему) через Y ошибочен; язык не определяет, что происходит в результате такого доступа.

Примечание. Согласно правилам видимости настраиваемая процедура UNCHECKED_DEALLOCATION не видима в компилируемом модуле, если только ее имя не указано в спецификаторе совместности этого компилируемого модуля.

Если X указывает на объект задачного типа, то вызов СВОБОДНО (X) никак не влияет на задачу, указанную значением этого объекта Это же относится и к любому подкомпоненту задачного типа объекта X.

13.10.2. Неконтролируемое преобразование типов

Неконтролируемое преобразование типа можно осуществить вызовом функции, полученной конкретизацией настраиваемой функции UNCHECKED_CONVERSION.

Неконтролируемое преобразование типа состоит в возврате значения параметра в качестве значения целевого типа, т. е. поразрядное изображение, определяющее исходное значение, возвращается неизменным, как поразрядное изображение значения целевого типа. Реализация может наложить ограничения на неконтролируемое преобразование типа, например, ограничения, зависящие от предполагаемых размеров объектов исходного и целевого типов. Такие ограничения должны быть отражены в руководстве по реализации в соответствии с обязательным приложением 4.

При использовании неконтролируемых преобразований типов сам программист несет ответственность за сохранность свойств, гарантируемых языком для объектов целевого типа. Программы, нарушающие их свойства при неконтролируемых преобразованиях, являются ошибочными.

Примечание. Согласно правилам видимости настраиваемая функция UNCHECKED_CONVERSION не видима в компилируемом модуле, если она не упомянута в спецификаторе совместности этого компилируемого модуля.

14. ВВОД-ВЫВОД

Ввод-вывод в языке обеспечивается предопределенными пакетами. Настраиваемые пакеты SEQUENTIAL_IO и DIRECT_IO определяют операции

ввода-вывода, которые применимы для файлов с элементами данного типа. В пакете TEXT_IO даны дополнительные операции ввода-вывода текстов. В пакете IO_EXCEPTIONS определены исключения, необходимые для трех указанных пакетов. Наконец, пакет LOW_LEVEL_IO позволяет осуществлять непосредственное управление периферийными устройствами.

14.1. **Внешние файлы и файловые объекты**

Значения, вводимые из внешнего для программы окружения или выводимые в это окружение, размещаются во *внешних файлах*. Внешним файлом *может быть нечто внешнее по отношению к программе, которая может* произвести читаемое значение или получить записываемое. Внешний файл идентифицируется строкой (*именем*). Вторая строка (*форма*) задает дополнительные системо-зависимые характеристики, которые могут быть сопоставлены с файлом, например, физическая организация или права доступа. Соглашения об интерпретации таких строк должны быть приведены в руководстве по реализации в соответствии с обязательным приложением 4.

Операции ввода и вывода выражены операциями над объектами некоторого *файлового типа,* а не непосредственно в терминах внешних файлов. Далее в этой главе термин *файл* будет всегда использоваться для ссылки на объект файлового типа; в остальных случаях будет использоваться термин *внешний файл.* Значения, передаваемые данному файлу, должны быть все одного и того же типа.

Ввод и вывод для последовательных файлов из элементов некоторого типа определены настраиваемым пакетом SEQUENTIAL_IO. Общая структура этого пакета дана ниже.

```
with IO_EXCEPTIONS;
generic
  type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is
  type FILE_TYPE is limited private;
  type FILE_MODE is (IN_FILE, OUT_FILE);
  . . .
  procedure OPEN (FILE: in out FILE_TYPE; . . .);
  . . .
  procedure READ (FILE: in FILE_TYPE; ITEM: out ELEMENT_TYPE) ;
  procedure WRITE (FILE: in FILE_TYPE; ITEM: in ELEMENT_TYPE);
  . . .
end SEQUENTIAL_IO;
```

Для определения последовательного ввода-вывода элементов данного типа должна быть описана конкретизация этого настраиваемого модуля с фактическим параметром данного типа. Результат настройки содержит описание файлового типа (названного FILE_TYPE) для файлов с такими элементами, а также операции над этими файлами; например, процедуры OPEN, READ и WRITE.

Ввод-вывод для файлов прямого доступа определен аналогичным способом в настраиваемом пакете DIRECT_IO. Ввод-вывод в текстовой форме определен в (ненастраиваемом) пакете TEXT_IO.

До выполнения ввода или вывода как операции над файлом должна быть установлена связь файла с внешним файлом. Когда такая связь установлена, файл называется *открытым,* в противном случае — *закрытым.*

В языке не определено, что происходит с внешними файлами после завершения работы главной программы (в частности, если соответствующие файлы не были закрыты). Результат выполнения ввода-вывода над ссылочными типами зависит от реализации.

Открытый файл имеет *текущий вид*, который является значением одного из перечислимых типов:

```
type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE); - - для DIRECT_IO
type FILE_MODE is (IN_FILE, OUT_FILE); - - для SEQUENTIAL_IO и TEXT_IO
```

Эти значения соответствуют случаям, когда можно осуществить либо только чтение, либо чтение и запись, либо только запись. Вид файла может быть изменен.

Некоторые из операций управления файлами являются общими для всех трех пакетов ввода-вывода. Эти операции над последовательными файлами и файлами прямого доступа описаны в разд. 14.2.1. Дополнительные особенности ввода-вывода текстов описаны в разд. 14.3.1.

В пакете IO_EXCEPTIONS определены все исключения, которые могут быть возбуждены при вызове подпрограммы ввода-вывода; ситуации их возбуждения описаны либо при описании подпрограмм ввода и вывода (и в разд. 14.4), либо в руководстве по реализации в соответствии с обязательным приложением 4 для ошибочных ситуаций, зависящих от реализации.

Примечание. Каждая конкретизация настраиваемых пакетов SEQUENTIAL_IO и DIRECT_IO задает разные типы FILE_TYPE; уникальным является тип FILE_TYPE в пакете TEXT_IO.

Устройство с двусторонней связью часто может быть промоделировано двумя файлами с последовательным доступом, связанными с этим устройством; один – вида IN_FILE, а другой – вида OUT_FILE. Реализация может ограничить число файлов, которые могут быть связаны с данным внешним файлом. В этом случае результат такого разделения внешнего файла несколькими объектами файлового типа зависит от реализации.

14.2. **Файлы последовательного и прямого доступа**

Для внешних файлов определены *последовательный доступ* и *прямой доступ*. В настраиваемых пакетах SEQUENTIAL_IO и DIRECT_IO описаны соответствующие файловые типы и связанные с ними операции. Объект файлового типа, используемый для последовательного доступа, называется *последовательным файлом*, а используемый для прямого доступа – *прямым файлом*.

При последовательном доступе файл рассматривается как последовательность значений, которые передаются в порядке их поступления (от программы или из окружения). Если файл открывается, то передача начинается с начала файла.

При прямом доступе файл рассматривается как набор элементов, занимающих последовательные позиции в линейном порядке; значение может быть передано в элемент файла (или из него), находящийся в любой выбранной позиции. Позиция элемента задается его *индексом*, который является положительным числом определяемого реализацией целого типа COUNT.

Индекс первого элемента в файле (если он есть) равен единице; индекс последнего элемента (если он есть) называется *текущим размером;* текущий размер файла, не содержащего ни одного элемента, равен нулю. Текущий размер – это характеристика внешнего файла.

Открытый прямой файл имеет *текущий индекс,* который будет использован операцией для чтения или записи. При открытии прямого файла значение текущего индекса устанавливается равным единице. Текущий индекс прямого файла – это характеристика не внешнего файла, а связанного с ним объекта файлового типа.

Для прямых файлов допустимы все три вида файла. Для последовательных файлов допустимы только виды IN_FILE и OUT_FILE.

14.2.1. Управление файлами

В этом разделе описаны процедуры и функции, предназначенные для управления внешними файлами; их описания повторены в каждом из трех пакетов последовательного, прямого и текстового ввода-вывода. Процедуры CREATE, OPEN и RESET при вводе-выводе текстов имеют дополнительные эффекты, описанные в разд. 14.3.1.

```
procedure CREATE (FILE: in out FILE_TYPE;
                  MODE: in FILE_MODE: = вид_по_умолчанию;
                  NAME: in STRING: = " ";
                  FORM: in STRING: = " ");
```

Устанавливает новый внешний файл с данным именем и формой и связывает его с данным файлом (FILE). После этого данный файл открывается. Текущий вид данного файла устанавливается в заданный вид доступа (MODE). По умолчанию для последовательного и текстового ввода-вывода устанавливается вид OUT_FILE, а для прямого ввода-вывода – INOUT_FILE. Для прямого доступа размер созданного файла зависит от реализации. Пустая строка с именем NAME задает внешний файл, который не доступен после окончания главной программы (временный файл). Пустая строка для формы FORM задает параметры по умолчанию, определяемые реализацией для внешнего файла.

Если данный файл уже открыт, то возбуждается исключение STATUS_ERROR. Если указанная в качестве параметра NAME строка не допускает идентификацию внешнего файла, то возбуждается исключение NAME_ERROR. Если для заданного вида файла окружение не может обеспечить создание внешнего файла с заданными именем и формой, то возбуждается исключение USE_ERROR (в отсутствие возбуждения исключения NAME_ERROR)

```
procedure OPEN (FILE: in out FILE_TYPE;
                MODE: in FILE_MODE;
                NAME: in STRING;
                FORM: in STRING: = " ");
```

Связывает данный файл с существующим внешним файлом, имеющим данные имя и форму, а текущий вид данного файла устанавливается параметром MODE. Данный файл открывается.

Если данный файл уже открыт, то возбуждается исключение STATUS_ERROR. Если строка, заданная параметром NAME, не допускает

идентификацию внешнего файла, то возбуждается исключение NAME_ERROR; в частности, это исключение возбуждается, если внешнего файла с указанным именем не существует. Если для заданного вида файла окружение не может обеспечить открытие внешнего файла с данными именем и формой, то возбуждается исключение USE_ERROR (в отсутствие возбуждения исключения NAME_ERROR).

procedure CLOSE (FILE: **in out** FILE_TYPE);

Уничтожает связь между данным файлом и соответствующим ему внешним файлом. Данный файл закрывается.

Если данный файл не открыт, то возбуждается исключение STATUS_ERROR.

procedure DELETE (FILE: **in out** FILE_TYPE);

Уничтожает внешний файл, связанный с данным файлом. Данный файл закрывается, внешний файл прекращает существование.

Если данный файл не открыт, то возбуждается исключение STATUS_ERROR. Если уничтожение внешнего файла не может быть обеспечено окружением, то возбуждается исключение USE_ERROR (все такие случаи должны быть описаны в руководстве по реализации в соответствии с обязательным приложением 4).

procedure RESET (FILE: **in out** FILE_TYPE; MODE: **in** FILE_MODE);
procedure RESET (FILE: **in out** FILE_TYPE);

Устанавливает данный файл в состояние, позволяющее возобновить чтение или запись значений его элементов с начала файла; в частности, для прямого доступа это означает, что текущий индекс становится равным единице. Если задан параметр MODE, то в соответствии с ним устанавливается текущий вид данного файла.

Если файл не открыт, то возбуждается исключение STATUS_ERROR. Если для внешнего файла окружение не может осуществить возврат к началу файла или установку данного вида, то возбуждается исключение USE_ERROR.

function MODE (FILE: **in** FILE_TYPE) **return** FILE_MODE;

Возвращает текущий вид данного файла.

Если файл не открыт, то возбуждается исключение STATUS_ERROR.

function NAME (FILE: **in** FILE_TYPE) **return** STRING;

Возвращает строку, которая однозначно идентифицирует внешний файл, связанный с данным файлом (она может быть использована в операции OPEN). Если окружение допускает альтернативные спецификации имени (например, сокращения), то возвращаемая функцией строка обязана соответствовать полной спецификации имени.

Если данный файл не открыт, то возбуждается исключение STATUS_ERROR.

function FORM (FILE: **in** FILE_TYPE) **return** STRING;

Возвращает строку, определяющую форму внешнего файла, связанного в этот момент с данным файлом. Если окружение допускает альтернативные спецификации форм (например, сокращения, использующие возможности по умолчанию), то возвращаемая функцией строка обязана соответ-

ствовать полной спецификации (т. е. она обязана явно содержать все выбранные возможности, включая возможности по умолчанию).

Если данный файл не открыт, то возбуждается исключение STATUS_ERROR.

```
function IS_OPEN (FILE: in FILE_TYPE) return BOOLEAN;
```

Если файл открыт (т. е. связан с внешним файлом), то возвращает значение TRUE, в противном случае – FALSE.

14.2.2. Последовательный ввод-вывод

В этом разделе описаны операции для последовательного ввода и вывода. В случае применения любой из этих операций к закрытому файлу возбуждается исключение STATUS_ERROR.

```
procedure READ (FILE: in FILE_TYPE; ITEM: out ELEMENT_TYPE);
```

Оперирует над файлом вида IN_FILE. Читает элемент данного файла и возвращает значение этого элемента через параметр ITEM.

Если вид файла не IN_FILE, то возбуждается исключение MODE_ERROR. Если из файла нельзя больше читать ни одного элемента, то возбуждается исключение END_ERROR. Если прочитанный элемент не может быть интерпретирован как значение типа ELEMENT_TYPE, то возбуждается исключение DATA_ERROR; однако для реализации допустимо опускать такую проверку в случае, если она слишком сложна.

```
procedure WRITE (FILE: in FILE_TYPE; ITEM: in ELEMENT_TYPE);
```

Оперирует над файлом вида OUT_FILE. Записывает в данный файл значение параметра ITEM.

Если вид файла не OUT_FILE, то возбуждается исключение MODE_ERROR. Если внешний файл уже заполнен до конца, то возбуждается исключение USE_ERROR.

```
function END_OF_FILE (FILE: in FILE_TYPE) return BOOLEAN;
```

Оперирует над файлом вида IN_FILE. Если из файла больше нельзя читать ни одного элемента, то возвращает значение TRUE; в противном случае – FALSE.

Если вид файла не IN_FILE, то возбуждается исключение MODE_ERROR.

14.2.3. Спецификация пакета последовательного ввода-вывода

```
with IO_EXCEPTIONS;
generic
  type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is
  type FILE_TYPE is limited private;
  type FILE_MODE is (IN_FILE, OUT_FILE);
-- управление файлами
  procedure CREATE (FILE: in out FILE_TYPE;
                    MODE: in FILE_MODE: = OUT_FILE;
                    NAME: in STRING: = " ";
                    FORM: in STRING: = " ");
  procedure OPEN (FILE: in out FILE_TYPE;
                  MODE: in FILE_MODE;
```

```
                    NAME: in STRING;
                    FORM: in STRING: = " ");
  procedure CLOSE (FILE: in out FILE_TYPE);
  procedure DELETE (FILE: in out FILE_TYPE);
  procedure RESET (FILE: in out FILE_TYPE;
                   MODE: in FILE_MODE);
  procedure RESET (FILE: in out FILE_TYPE);
  function MODE (FILE: in FILE_TYPE) return FILE_MODE;
  function NAME (FILE: in FILE_TYPE) return STRING;
  function FORM (FILE: in FILE_TYPE) return STRING;
  function IS_OPEN (FILE: in FILE_TYPE) return BOOLEAN;
  - - операции ввода и вывода
  procedure READ (FILE: in FILE_TYPE;
                  ITEM: out ELEMENT_TYPE);
  procedure WRITE (FILE: in FILE_TYPE;
                   ITEM: in ELEMENT_TYPE);
  function END_OF_FILE (FILE: in FILE_TYPE) return BOOLEAN;
  - - исключения
  STATUS_ERROR: exception renames IO_EXCEPTIONS. STATUS_ERROR;
  MODE_ERROR: exception renames IO_ EXCEPTIONS. MODE_ERROR;
  NAME_ERROR: exception renames IO_EXCEPTIONS. NAME_ERROR;
  USE_ERROR: exception renames IO_EXCEPTIONS. USE_ERROR;
  DEVICE_ERROR: exception renames IO_EXCEPTIONS. DEVICE_ERROR;
    END_ERROR: exception renames IO_EXCEPTIONS. END_ERROR;
    DATA_ERROR: exception renames IO_EXCEPTIONS. DATA_ERROR;
private
  - - зависит от реализации
end SEQUENTIAL_IO;
```

14.2.4. Прямой ввод - вывод

В этом разделе описаны операции для прямого ввода и вывода. При применении любой из этих операций над файлом, который не открыт, возбуждается исключение STATUS_ERROR.

```
procedure READ (FILE: in FILE_TYPE;
                ITEM: out ELEMENT_TYPE;
                FROM: in POSITIVE_COUNT);
procedure READ (FILE: in FILE_TYPE;
                ITEM: out ELEMENT_TYPE);
```

Оперирует над файлами вида IN_FILE или INOUT_FILE. Первая из подпрограмм предварительно устанавливает текущий индекс данного файла равным значению параметра FROM. Затем (для обеих подпрограмм) через параметр ITEM возвращает значение элемента файла, позиция которого задана текущим индексом файла; наконец, увеличивает текущий индекс на единицу.

Если вид данного файла – OUT_FILE, то возбуждается исключение MODE_ERROR. Если используемое при чтении значение индекса оказалось больше размера внешнего файла, то возбуждается исключение END_ERROR. Если прочитанный элемент нельзя интерпретировать как значение типа ELEMENT_TYPE, то возбуждается исключение DATA_ERROR; однако для реализации допустимо опускать такую проверку в случае, если она слишком сложна.

```
procedure WRITE (FILE: in FILE_TYPE;
                 ITEM: in ELEMENT_TYPE;
                 TO: in POSITIVE_COUNT);
```

```
procedure WRITE (FILE: in FILE_TYPE; ITEM: in ELEMENT_TYPE);
```

Оперирует над файлами вида INOUT_FILE или OUT_FILE. Первая из операций предварительно устанавливает индекс данного файла равным значению параметра TO. Затем (для обеих подпрограмм) элементу данного файла, позиция которого указана текущим индексом, присваивает значение параметра ITEM; наконец, увеличивает текущий индекс на единицу.

Если вид данного файла – IN_FILE, то возбуждается исключение MODE_ERROR. Если внешний файл заполнен до конца, то возбуждается исключение USE_ERROR.

```
procedure SET_INDEX (FILE: in FILE_TYPE; TO: in POSITIVE_COUNT);
```

Оперирует над файлом любого вида. Устанавливает текущий индекс данного файла равным значению параметра TO (которое может превышать текущий размер файла).

```
function INDEX (FILE: in FILE_TYPE) return POSITIVE_COUNT;
```

Оперирует над файлом любого вида. Возвращает текущий индекс данного файла.

```
function SIZE (FILE: in FILE_TYPE) return COUNT;
```

Оперирует над файлом любого вида. Возвращает текущий размер внешнего файла, связанного с данным файлом.

```
function END_OF_FILE (FILE: in FILE_TYPE) return BOOLEAN;
```

Оперирует над файлами вида IN_FILE или INOUT_FILE. Если значение текущего индекса больше размера внешнего файла, то возвращает значение TRUE; в противном случае – FALSE.

Если вид данного файла – OUT_FILE, то возбуждается исключение MODE_ERROR.

14.2.5. Спецификация пакета прямого ввода-вывода

```
with IO_EXCEPTIONS;
generic
  type ELEMENT_TYPE is private;
package DIRECT_IO is
  type FILE_TYPE is limited private;
  type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
  type COUNT is range 0. .определяется_реализацией:
    subtype POSITIVE_COUNT is COUNT range 1. .COUNT'LAST;
  -- управление файлами
  procedure CREATE (FILE: in out FILE_TYPE;
                    MODE: in FILE_MODE: = INOUT_FILE;
                    NAME: in STRING: = " ";
                    FORM: in STRING: = " ");
  procedure OPEN (FILE: in out FILE_TYPE;
                  MODE: in FILE_MODE;
                  NAME: in STRING;
                  FORM: in STRING: = " ");
  procedure CLOSE (FILE: in out FILE_TYPE);
  procedure DELETE (FILE: in out FILE_TYPE);
  procedure RESET (FILE: in out FILE_TYPE; MODE: in FILE_MODE);
  procedure RESET (FILE: in out FILE_TYPE);
  function MODE (FILE: in FILE_TYPE) return FILE_MODE;
```

```
function NAME (FILE: in FILE_TYPE) return STRING;
function FORM (FILE: in FILE_TYPE) return STRING;
function IS_OPEN (FILE: in FILE_TYPE) return BOOLEAN;
- - операции ввода и вывода
procedure READ (FILE: in FILE_TYPE;
                ITEM: out ELEMENT_TYPE;
                FROM: POSITIVE_COUNT);
procedure READ (FILE: in FILE_TYPE; ITEM: out ELEMENT_TYPE);
procedure WRITE (FILE: in FILE_TYPE;
                 ITEM: in ELEMENT_TYPE;
                 TO: POSITIVE_COUNT);
procedure WRITE (FILE: in FILE_TYPE; ITEM: in ELEMENT_TYPE);
procedure SET_INDEX (FILE: in FILE_TYPE; TO: in POSITIVE_COUNT);
function INDEX (FILE: in FILE_TYPE) return POSITIVE_COUNT;
function SIZE (FILE: in FILE_TYPE) return COUNT;
function END_OF_FILE (FILE: in FILE_TYPE) return BOOLEAN;
- - исключения
STATUS_ERROR: exception renames IO_EXCEPTIONS. STATUS_ERROR;
MODE_ERROR: exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR: exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR: exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR: exception renames IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR: exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR: exception renames IO_EXCEPTIONS.DATA_ERROR;
private
- - зависит от реализации
end DIRECT_IO;
```

14.3. Ввод-вывод текстов

В этом разделе описывается пакет TEXT_IO, который обеспечивает возможности ввода и вывода в удобной для читателя форме. Каждый файл читается или записывается последовательно посимвольно, символы последовательно группируются в строчки, последовательность строчек – в страницы. В разд. 14.3.10 приведена спецификация этого пакета.

Возможности управления файлами, описанные в разд. 14.2.1 и 14.2.2, применимы и для текстового ввода-вывода. Однако вместо процедур READ и WRITE используются процедуры GET и PUT, которые осуществляют ввод и вывод значений соответствующих типов для текстовых файлов. Эти значения передаются процедурами PUT и возвращаются процедурами GET через параметр ITEM. Существует несколько совмещенных процедур с такими именами, но с различными типами параметра ITEM. Процедуры GET анализируют вводимые последовательности символов как лексемы (см. гл. 2) и возвращают соответствующие значения; процедуры PUT выводят данные значения в виде соответствующих лексем. Процедуры GET и PUT могут также вводить и выводить отдельные символы, рассматриваемые не как лексемы, а как значения символьного типа.

Для числового и перечислимого типов параметра ITEM, помимо процедур PUT и GET, записывающих в текстовый файл или читающих из него, существуют аналогичные процедуры с параметром типа STRING. Эти процедуры производят точно такие же анализ и формирование последовательности символов, как и подобные им процедуры с файловым параметром.

Для всех процедур GET и PUT, оперирующих над текстовыми файлами, а также для многих других подпрограмм, существуют формы как с параметром файлового типа, так и без него. Каждая процедура GET оперирует над файлом ввода; каждая процедура PUT – над файлом вывода. Если файл не задан, то работа производится над файлом ввода по умолчанию или над файлом вывода по умолчанию.

В начале выполнения программы файлами ввода и вывода по умолчанию являются так называемые стандартный файл ввода и стандартный файл вывода. Эти файлы всегда открыты и имеют текущие виды IN_FILE и OUT_FILE соответственно, они связаны с двумя определяемыми реализацией внешними файлами. Существуют процедуры для замены текущего файла ввода по умолчанию и текущего файла вывода по умолчанию.

Логически текстовый файл представляет собой последовательность страниц, страница – последовательность строчек, а строчка – последовательность символов; конец строчки помечается *признаком конца строчки;* конец страницы помечается комбинацией *признака конца строчки,* за которым непосредственно следует *признак конца страницы;* конец файла помечается комбинацией следующих непосредственно друг за другом признака конца строчки, признака конца страницы и *признака конца файла.* Признаки конца генерируются во время вывода: либо при вызове специально предусмотренных для этого процедур, либо неявно – как составная часть других операций, например, когда для файла заданы ограничения длины строчки, длины страницы или оба эти ограничения.

Язык не определяет, что фактически представляют из себя признаки конца; это зависит от реализации. Хотя некоторые из описанных ниже процедур могут опознавать или сами генерировать признаки конца, которые не обязаны всегда быть реализованы именно как символы или последовательности символов. Пользователю безразлично, представлены ли в данной конкретной реализации признаки конца как символы (и, если да, то какие именно), так как он никогда явно не вводит и явно не выводит управляющие символы. Язык не определяет результаты ввода или вывода управляющих символов (кроме символа горизонтальной табуляции).

Символы строчки пронумерованы, начиная с единицы; номер символа называется *номером столбца.* Для признака конца строчки также определен номер столбца: его значение на единицу больше числа символов в строчке. Строчки страницы и страницы файла перенумерованы аналогично. *Текущий номер столбца* – это номер очередного (передаваемого) символа или признака конца строчки. *Текущий номер строчки* – это номер текущей строчки в текущей странице. *Текущий номер страницы* – это номер текущей страницы в файле. Все эти номера являются значениями подтипа POSITIVE_COUNT типа COUNT (нулевое значение типа COUNT по соглашению используется для специальных целей).

```
type COUNT is range 0. . определяется_реализацией;
subtype POSITIVE_COUNT is COUNT range 1. .COUNT'LAST;
```

Для файла вывода могут быть заданы *максимальная длина строчки* и *максимальная длина страницы*. Если они заданы, а значение не помещается на текущей строчке, то автоматически до вывода значения будет начата новая строчка; если, далее, это новая строчка не может быть размещена на текущей странице, то автоматически до вывода будет начата новая страница. Имеются функции для определения максимальной длины строчки и максимальной длины страницы. После открытия файла вида OUT_FILE оба эти значения равны нулю; это значит, что длина страницы и длина строчки считаются неограниченными. (Следовательно, весь файл вывода состоит из одной строчки, если только не используются подпрограммы явного управления структурой строчек и страниц файла.) Для этих целей служит константа UNBOUNDED.

14.3.1. Управление файлами

Для текстовых файлов допустимы только виды IN_FILE и OUT_FILE. К ним также применимы приведенные в разд. 14.2.1 подпрограммы управления внешними файлами, а также приведенная в разд. 14.2.2 функция END_OF_FILE для последовательного ввода-вывода. Имеется также вариант функции END_OF_FILE, который выдает результат для текущего файла ввода по умолчанию. Указанные процедуры для текстовых файлов характеризуются следующим:

- Процедуры CREATE и OPEN: после открытия файла вида OUT_FILE длина страницы и длина строчки не ограничены (имеют по соглашению значение 0). После открытия файла вида IN_FILE или OUT_FILE текущие номера столбца, строчки и страницы устанавливаются равными единице.
- Процедура CLOSE: если файл имеет текущий вид OUT_FILE, а текущая страница еще не завершена, то результат эквивалентен вызову подпрограммы NEW_PAGE; затем выводится признак конца файла.
- Процедура RESET: если файл имеет текущий вид OUT_FILE, а текущая страница еще не завершена, то результат эквивалентен вызову подпрограммы NEW_PAGE; затем выводится признак конца файла. Если новый вид файла – OUT_FILE, то длина строчки и длина страницы становятся неограниченными. Для всех видов файла текущие номера столбца, строчки и страницы устанавливаются равными единице.

При попытке изменить вид текущего файла ввода по умолчанию или текущего файла вывода по умолчанию процедурой PESET возбуждается исключение MODE_ERROR.

14.3.2. Файлы ввода и вывода по умолчанию

Следующие подпрограммы служат для управления файлами по умолчанию, используемыми при отсутствии параметра-файла в процедурах PUT, GET или в других, описанных ниже, операциях текстового ввода-вывода.

procedure SET_INPUT (FILE: **in** FILE_TYPE);

Оперирует над файлом вида IN_FILE. Устанавливает текущим файлом ввода по умолчанию файл, заданный параметром FILE.

Если данный файл не открыт, то возбуждается исключение STATUS_ERROR. Если вид данного файла отличен от IN_FILE, то возбуждается исключение MODE_ERROR.

procedure SET_OUTPUT (FILE: **in** FILE_TYPE);

Оперирует над файлом вида OUT_FILE. Устанавливает текущим файлом вывода по умолчанию файл, заданный параметром FILE.

Если данный файл не открыт, то возбуждается исключение STATUS_ERROR. Если вид данного файла отличен от OUT_FILE, то возбуждается исключение MODE_ERROR.

function STANDARD_INPUT **return** FILE_TYPE;

Возвращает стандартный файл ввода (см. разд. 14.3).

function STANDARD_OUTPUT **return** FILE_TYPE;

Возвращает стандартный файл вывода (см. разд. 14.3).

function CURRENT_INPUT **return** FILE_TYPE;

Возвращает текущий файл ввода по умолчанию.

function CURRENT_OUTPUT **return** FILE_TYPE;

Возвращает текущий файл по умолчанию.

Примечание. Стандартные файлы ввода и вывода не могут быть открыты, закрыты, переустановлены и уничтожены, потому что в соответствующих процедурах параметр FILE имеет вид **in out**.

14.3.3. Спецификации длин строчек и страниц

Описанные в этом разделе подпрограммы связаны со структурой строчек и страниц файла вида OUT_FILE. Они оперируют либо над файлом, заданным первым параметром, либо, при отсутствии такого параметра, над текущим файлом вывода по умолчанию. Эти подпрограммы служат для вывода текста с заданными максимальными длинами строчки или страницы. В этих случаях признаки конца строчки и страницы выводятся неявно и автоматически, по мере необходимости. Когда длины строчки и страницы не ограничены (т. е. когда они имеют по соглашению значение нуль), как в случае заново открытого файла, новые строчки или страницы можно начать лишь в результате явного вызова соответствующих подпрограмм.

Во всех случаях, если заданный файл не открыт, то возбуждается исключение STATUS_ERROR; если вид этого файла отличен от OUT_FILE, то возбуждается исключение MODE_ERROR.

procedure SET_LINE_LENGTH (FILE: **in** FILE_TYPE; TO: **in** COUNT);
procedure SET_LINE_LENGTH (TO: **in** COUNT);

Устанавливает максимальную длину строчки заданного файла вывода равной числу символов, задаваемому значением параметра TO. Нулевое значение параметра TO задает неограниченную длину строчки.

Если заданное значение длины строчки не подходит для связанного внешнего файла, то возбуждается исключение USE_ERROR.

procedure SET_PAGE_LENGTH (FILE: **in** FILE_TYPE; TO: **in** COUNT);
procedure SET_PAGE_LENGTH (TO: **in** COUNT);

Устанавливает максимальную длину страницы заданного файла вывода равной числу строчек, задаваемому значением параметра TO. Нулевое значение параметра TO задает неограниченную длину страницы.

Если данное значение длины страницы не подходит для связанного внешнего файла, то возбуждается исключение USE_ERROR.

function LINE_LENGTH (FILE: **in** FILE_TYPE) **return** COUNT;
function LINE_LENGTH **return** COUNT;

Возвращает максимальную длину строчки, установленную для заданного файла вывода, или нуль, если длина строчки не ограничена.

function PAGE_LENGTH (FILE: **in** FILE_TYPE) **return** COUNT;
function PAGE_LENGTH **return** COUNT;

Возвращает максимальную длину страницы, установленную для данного файла, или нуль, если длина страницы не ограничена.

14.3.4. Операции над столбцами, строчками и страницами

Описанные в этом разделе подпрограммы предназначены для явного управления структурой строчек и страниц файла; они оперируют либо над файлом, заданным первым параметром, либо, при отсутствии такого параметра-файла, над текущим файлом (ввода или вывода) по умолчанию. Если используемый в этих подпрограммах файл не открыт, то возбуждается исключение STATUS_ERROR.

procedure NEW_LINE (FILE: **in** FILE_TYPE;
SPACING: **in** POSITIVE_COUNT: = 1);
procedure NEW_LINE (SPACING: **in** POSITIVE_COUNT: = 1);

Оперирует над файлом вида OUT_FILE.

Если SPACING равно единице, то выводит признак конца строчки, а текущий номер столбца устанавливается равным единице. Затем увеличивает текущий номер строчки на единицу, кроме случая, когда текущий номер строчки уже был равен или превышал максимальную длину страницы; в этом случае сначала выводит признак конца страницы, увеличивает затем текущий номер страницы на единицу, а текущий номер строчки устанавливает равным единице.

Если SPACING больше единицы, то указанные действия повторяются SPACING раз.

Если вид файла не OUT_FILE, то возбуждается исключение MODE_ERROR.

procedure SKIP_LINE (FILE: **in** FILE_TYPE;
SPACING: **in** POSITIVE_COUNT: = 1);
procedure SKIP_LINE (SPACING: **in** POSITIVE_COUNT: = 1);

Оперирует над файлом вида IN_FILE.

Если SPACING равен единице, то считывает из файла и пропускает (игнорирует) все символы до признака конца строчки, а текущий номер столбца устанавливает равным единице. Если непосредственно за признаком конца строчки не следует признак конца страницы, то увеличивает текущий номер строчки на единицу. В противном случае, когда за признаком конца строчки непосредственно следует признак конца страницы, пропускает признак конца страницы, увеличивает текущий номер страницы на единицу, а текущий номер строчки устанавливает равным единице.

Если SPACING больше единицы, указанные действия повторяются SPACING раз.

Если вид файла не IN_FILE, то возбуждается исключение MODE_ERROR. При попытке прочитать признак конца файла возбуждается исключение END_ERROR.

```
function END_OF_LINE (FILE: in FILE_TYPE) return BOOLEAN;
function END_OF_LINE return BOOLEAN;
```

Оперирует над файлом вида IN_FILE. Возвращает значение TRUE, если текущий элемент файла – это признак конца строчки или признак конца файла; в противном случае возвращает значение FALSE.

Если вид файла не IN_FILE, то возбуждается исключение MODE_ERROR.

```
procedure NEW_PAGE (FILE: in FILE_TYPE);
procedure NEW_PAGE;
```

Оперирует над файлом вида OUT_FILE. Если текущая строчка не завершена или текущая страница пустая (т. е. текущие номера строчки или столбца оба равны единице), то выводит признак конца строчки. Затем выводит признак конца страницы, который завершает текущую страницу. Увеличивает номер текущей страницы на единицу, а текущие номера столбца и строчки устанавливает равными единице.

Если вид файла не OUT_FILE, то возбуждается исключение MODE_ERROR.

```
procedure SKIP_PAGE (FILE: in FILE_TYPE);
procedure SKIP_PAGE;
```

Оперирует над файлами вида IN_FILE. Из файла читает и пропускает (игнорирует) все символы и признаки конца строчки до признака конца страницы. Увеличивает текущий номер страницы на единицу, текущие номера столбца и строчки устанавливает равными единице.

Если вид файла не IN_FILE, то возбуждается исключение MODE_ERROR. При попытке прочитать признак конца файла возбуждается исключение END_ERROR.

```
function END_OF_PAGE (FILE: in FILE_TYPE) return BOOLEAN;
function END_OF_PAGE return BOOLEAN;
```

Оперирует над файлом вида IN_FILE. Возвращает значение TRUE, если очередными элементами файла является последовательность из признаков конца строчки и страницы или если очередным элементом является признак конца файла; в противном случае возвращает значение FALSE.

Если вид файла отличен от IN_FILE, то возбуждается исключение MODE_ERROR.

```
function END_OF_FILE (FILE: in FILE_TYPE) return BOOLEAN;
function END_OF_FILE return BOOLEAN;
```

Оперирует над файлом вида IN_FILE. Возвращает значение TRUE, если очередным элементом файла является признак конца файла или последовательность из признаков конца строчки, страницы и файла; в противном случае возвращает значение FALSE.

Если вид файла отличен от IN_FILE, то возбуждается исключение MODE_ERROR.

Следующие подпрограммы предназначены для управления текущей позицией чтения или записи в файл. Во всех этих подпрограммах в качестве файла по умолчанию используется текущий файл вывода.

```
procedure SET_COL (FILE: in FILE_TYPE;
                   TO: in POSITIVE_COUNT);
procedure SET_COL (TO: in POSITIVE_COUNT);
```

Для файла вида OUT_FILE.

Если значение параметра TO больше текущего номера столбца, то выводит пробелы, причем после вывода каждого пробела текущий номер столбца увеличивает на единицу. Это повторяется до тех пор, пока текущий номер столбца не станет равным значению параметра TO. Если значение параметра TO было равно текущему номеру столбца, то никаких действий не производит. Если значение параметра TO меньше текущего номера столбца, то сначала выполняет действия, эквивалентные вызову процедуры NEW_LINE (SPACING = 1), затем выводит (TO – 1) пробелов, и текущий номер столбца устанавливает равным значению параметра TO.

Если при ограниченной длине строчки (т. е. LINE_LENGTH для этого файла имеет ненулевое значение) значение параметра TO оказалось больше LINE_LENGTH, то возбуждается исключение LAYOUT_ERROR.

Для файла вида IN_FILE.

Читает и пропускает (игнорирует) отдельные символы, признаки конца строчки и страницы до тех пор, пока номер столбца очередного, подлежащего чтению символа, не станет равным значению параметра TO. Если текущий номер столбца с самого начала равен этому значению, то никаких действий не производит. При передаче каждого символа или признака конца должным образом корректирует текущие номера столбца, строчки и страницы, как при работе процедуры GET (см. разд. 14.3.5). (Короткие строчки будут пропущены целиком, пока не встретится строчка, содержащая символ в указанной позиции от начала строчки.)

При попытке чтения признака конца файла возбуждается исключение END_ERROR.

```
procedure SET_LINE (FILE: in FILE_TYPE;
                    TO: in POSITIVE_COUNT);
procedure SET_LINE (TO: in POSITIVE_COUNT);
```

Для файла вида OUT_FILE.

Если значение параметра TO больше текущего номера строчки, то эта процедура эквивалентна повторным вызовам NEW_LINE (SPACING = 1) до тех пор, пока текущий номер строчки не станет равным значению параметра TO. Если значение параметра TO было равно текущему номеру строчки, то никаких действий не производит. Если значение параметра TO меньше текущего номера строчки, то результат вызова процедуры эквивалентен вызову NEW_PAGE, за которым следует вызов NEW_LINE с параметром SPACING, равным (TO – 1).

Если при ограниченной длине страницы (т. е. PAGE_LENGTH для этого файла имеет ненулевое значение) значение параметра TO оказалось больше PAGE_LENGTH, то возбуждается исключение LAYOUT_ERROR.

Для файла вида IN_FILE.

Результат эквивалентен повторным вызовам процедуры SKIP_LINE (SPACING = 1) до тех пор, пока текущий номер строчки не примет значение параметра TO. Если текущий номер строчки с самого начала был равен значению параметра TO, то никаких действий не производит. (Короткие

страницы будут пропущены целиком, пока не встретится страница, содержащая строчку в указанной позиции от начала страницы.)

При попытке чтения признака конца файла возбуждается исключение END_ERROR.

function COL (FILE: **in** FILE_TYPE) **return** POSITIVE_COUNT;
function COL **return** POSITIVE_COUNT;

Возвращает текущий номер столбца.

Если этот номер больше значения COUNT'LAST, то возбуждается исключение LAYOUT_ERROR.

function LINE (FILE: **in** FILE_TYPE) **return** POSITIVE_COUNT;
function LINE **return** POSITIVE_COUNT;

Возвращает текущий номер строчки.

Если этот номер больше значения COUNT'LAST, то возбуждается исключение LAYOUT_ERROR.

function PAGE (FILE: **in** FILE_TYPE) **return** POSITIVE_COUNT;
function PAGE **return** POSITIVE_COUNT;

Возвращает текущий номер страницы.

Если этот номер больше значения COUNT'LAST, то возбуждается исключение LAYOUT_ERROR.

Номера столбца, строчки или страницы, вообще говоря, могут превысить значение COUNT'LAST (в результате ввода или вывода достаточно большого числа символов, строчек или страниц). При этом никакое исключение не возбуждается. Однако при вызове функций COL, LINE или PAGE, если соответствующий номер оказался больше COUNT'LAST, то возбуждается исключение LAYOUT_ERROR.

Примечание. Признак конца страницы пропускается, если пропускается предшествующий ему признак конца строчки. Реализация может представить последовательность из таких двух признаков конца одним символом, при условии, что он будет распознаваться при вводе.

14.3.5. Процедуры обмена

В разд. 14.3.5–14.3.10 описаны процедуры GET и PUT для элементов типов CHARACTER, STRING, числового и перечислимого. В данном разделе описаны возможности этих процедур, общие для большинства таких типов. Процедуры GET и PUT для элементов типов CHARACTER и STRING передают отдельные символьные значения, а для числовых и перечислимых типов передают лексемы.

Первым параметром всех процедур GET и PUT является файл. Если он опущен, то подразумевается, что используется текущий файл (ввода или вывода) по умолчанию. Каждая процедура GET оперирует над файлом вида IN_FILE. Каждая процедура PUT оперирует над файлом вида OUT_FILE.

Все процедуры GET и PUT меняют для заданного файла текущие номера столбца, строчки и страницы: каждая передача символа увеличивает на единицу текущий номер столбца. Каждый вывод признака конца строчки устанавливает текущий номер столбца равным единице и добавляет единицу к текущему номеру строчки. Каждый вывод признака конца страницы устанавливает текущие номера столбца и строчки равными единице и до-

бавляет единицу к текущему номеру страницы. При вводе каждый признак конца строчки устанавливает текущий номер столбца равным единице и добавляет единицу к текущему номеру строчки; каждый признак конца страницы устанавливает текущие номера столбца и строчки равными единице и добавляет единицу к текущему номеру страницы. Аналогичным образом определяется семантика процедур GET_LINE, PUT_LINE и SET_COL.

Некоторые процедуры GET и PUT для числовых и перечислимых типов имеют параметры, задающие *формат*, который указывает длины полей; эти параметры принадлежат неотрицательному подтипу FIELD типа INTEGER.

Ввод-вывод значений перечислимых типов использует синтаксис соответствующих лексем. Любая процедура GET для перечислимого типа сначала пропускает все ведущие пропуски или признаки концов строчки и страницы; *пропуск* – это символ пробела или символ горизонтальной табуляции. Затем символы вводятся до тех пор, пока введенная последовательность является лексемой, соответствующей идентификатору или символьному литералу (в частности, ввод прекращается при достижении признака конца строчки). Символ или признак конца строчки, вызвавшие прекращение ввода, остаются доступными для следующего ввода.

Процедуры GET для числовых типов имеют параметр WIDTH, задающий формат. Если он имеет нулевое значение, то процедура GET выполняется так же, как для перечислимых типов, но вместо синтаксиса литералов перечисления используется синтаксис числовых литералов. При ненулевом значении параметра WIDTH вводится ровно WIDTH символов или, если ранее встретился признак конца строчки, то все символы до признака конца строчки; в это число включаются и все ведущие пропуски. Для числовых литералов используется расширенный синтаксис, в котором допускается знак числа (но не пропуски или признаки конца строчки или страницы внутри литерала).

Любая процедура PUT для элемента числового или перечислимого типов выводит значение элемента соответственно как числовой литерал, идентификатор или символьный литерал. Перед ними могут быть выведены пробелы, если этого требуют параметры формата WIDTH или FORE (это описано ниже), и для отрицательных значений – знак минус; в случае перечислимого типа пробелы выводятся не перед литералом, а после него. Если формат в процедуре PUT задает недостаточную ширину, то он игнорируется.

Следующие две ситуации могут возникнуть при выполнении процедуры PUT для числового или перечислимого типов в случае ограниченной длины строчки используемого файла вывода (т. е. длина строчки имеет ненулевое значение). Если число выводимых символов не превышает максимальной длины строчки, но при выводе этих символов, начиная с текущего столбца, они не помещаются в текущей строчке, то перед их выводом выполняются действия, эквивалентные вызову NEW_LINE с параметром SPACING, равным единице. Если же число выводимых символов больше максимальной длины строчки, то возбуждается исключение LAYOUT_ERROR, при этом символы не выводятся.

Если используемый в процедурах GET, GET_LINE, PUT и PUT_LINE файл не открыт, то возбуждается исключение STATUS_ERROR. Если в процедурах GET и GET_LINE вид используемого файла отличен от IN_FILE или в процедурах PUT и PUT_LINE вид используемого файла отличен от OUT_FILE, то возбуждается исключение MODE_ERROR.

В процедуре GET при попытке пропуска признака конца файла возбуждается исключение END_ERROR. Если вся введенная процедурой GET последовательность символов не является лексемой соответствующего типа, то возбуждается исключение DATA_ERROR; в частности, оно возбуждается, если не было введено ни одного символа; для числового типа, если был введен знак, то это правило относится к следующему за ним числовому литералу. В случае процедуры PUT, выводящий элемент типа STRING, если длина строки файла недостаточна для вывода заданного элемента, то возбуждается исключение LAYOUT_ERROR.

Примеры:

В примерах этого раздела и в разд. 14.3.7 и 14.3.8 кавычки и строчная буква *b* не вводятся и не выводятся; они даны только для того, чтобы показать расположение и пробелы.

```
K: INTEGER;
...
GET(K);
-- символы на входе, вводимая последовательность, значение K
-- bb – 12535b        – 12535          – 12535
-- bb 12_535E1b       12_535E1         125350
-- bb 12_535E;        12_535E          (нет) Возбуждено DATA_ERROR
```

Пример игнорирования параметра ширины:

```
PUT (ITEM => – 23, WIDTH => 2); -- "–23"
```

14.3.6. Ввод-вывод символов и строк

Для элемента типа CHARACTER определены следующие процедуры:

```
procedure GET (FILE: in FILE_TYPE; ITEM: out CHARACTER);
procedure GET (ITEM: out CHARACTER);
```

В заданном файле ввода процедуры после пропуска признаков конца строчки и страницы читают следующий за ними символ; значение этого символа возвращают параметру ITEM вида out.

При попытке пропустить признак конца файла возбуждается исключение END_ERROR.

```
procedure PUT (FILE: in FILE_TYPE; ITEM: in CHARACTER);
procedure PUT (ITEM: in CHARACTER);
```

Если длина строчки заданного файла вывода ограничена (т. е. не равна нулю по соглашению), а текущий номер столбца превышает эту длину, то выполняют действия, эквивалентные вызову процедуры NEW_LINE с параметром SPACING, равным единице. Затем в любом случае в файл выводят заданный символ.

Для элемента типа STRING определены следующие процедуры:

```
procedure GET (FILE: in FILE_TYPE; ITEM: out STRING);
procedure GET (ITEM: out STRING);
```

Определяют длину (число символов) данной строки; затем для последовательных символов строки соответствующее число раз выполняют операцию GET (в частности, для пустой строки никаких действий не выполняют).

procedure PUT (FILE: **in** FILE_TYPE; ITEM: **in** STRING);
procedure PUT (ITEM: **in** STRING);

Определяют длину (число символов) данной строки; затем для последовательных символов строки соответствующее число раз выполняют операцию PUT (в частности, для пустой строки никаких действий не выполняют).

procedure GET_LINE (FILE: **in** FILE_TYPE; ITEM: **out** STRING; LAST: **out** NATURAL);
procedure GET_LINE (ITEM: **out** STRING; LAST: **out** NATURAL);

Заменяют последовательные символы, содержащиеся в указанной строке, символами, читаемыми из заданного файла. Чтение заканчивается при достижении конца строчки файла; в этом случае выполняют действия, эквивалентные вызову процедуры SKIP_LINE с параметром SPACING, равным единице. Чтение также заканчивается и при достижении конца строки, заданной параметром ITEM. Символы, которые не были заменены, остаются не определенными.

Если символы прочитаны, то в параметр LAST возвращается индекс последнего замененного символа так, что индексированный компонент ITEM (LAST) – это значение последнего замененного символа (индекс первого замененного символа равен атрибуту ITEM'FIRST). Если не было прочитано ни одного символа, то в LAST выдается значение индекса на единицу меньше атрибута ITEM' FIRST.

При попытке пропустить признак конца файла возбуждается исключение END_ERROR.

procedure PUT_LINE (FILE: **in** FILE_TYPE; ITEM: **in** STRING);
procedure PUT_LINE (ITEM: **in** STRING);

Вызывают процедуру PUT для заданной строки, затем процедуру NEW_LINE с параметром SPACING, равным единице.

Примечание. Внешние кавычки строкового литерала, являющегося параметром процедуры PUT, не выводятся. Каждый сдвоенный символ кавычки, приведенный внутри такого литерала, выводится как один символ кавычки; это следует из правил для строковых литералов (см. разд. 2.6).

Строка, читаемая процедурой GET или записываемая процедурой PUT, в файле может занимать несколько строчек.

14.3.7. Ввод-вывод для целых типов

Описанные ниже процедуры определены в настраиваемом пакете INTEGER_IO. Он должен быть конкретизирован с соответствующим целым типом (указанным в спецификации параметром настройки NUM).

Значения выводятся в виде десятичных литералов или литералов с основанием, без подчеркиваний и порядка, с предшествующим знаком минус для отрицательных чисел. Формат, определяющий ширину поля (включая ведущие пробелы и знак минус), может быть задан необязательным па-

раметром WIDTH. Его значение принадлежит неотрицательному целому подтипу FIELD. Значения основания принадлежат целому подтипу NUMBER_BASE.

```
subtype NUMBER_BASE is INTEGER range 2. .16;
```

В процедурах вывода могут использоваться ширина поля и основание по умолчанию; они задаются переменными, описанными в настраиваемом пакете INTEGER_IO:

```
DEFAULT_WIDTH: FIELD: = NUM'WIDTH:
DEFAULT_BASE: NUMBER_BASE: = 10;
```

Определены следующие процедуры:

```
procedure GET (FILE: in FILE_TYPE; ITEM: out NUM; WIDTH: in FIELD: = 0);
procedure GET (ITEM: out NUM; WIDTH; in FIELD: = 0);
```

При нулевом значении параметра WIDTH пропускают все ведущие пропуски, признаки конца строчки и страницы, читают знак плюс или минус (если он есть), затем производят чтение в соответствии с синтаксисом целого литерала (он может быть литералом с основанием). При ненулевом значении WIDTH вводят ровно WIDTH символов или, если раньше встретится признак конца строки, то вводят лишь символы до этого признака конца (возможно, ни одного); в это количество включаются и все ведущие пропуски.

В параметр ITEM типа NUM возвращают значение, соответствующее введенной последовательности.

Если введенная последовательность не соответствует правилам синтаксиса или если полученное значение не принадлежит подтипу NUM, то возбуждается исключение DATA_ERROR.

```
procedure PUT (FILE: in FILE_TYPE; ITEM: in NUM;
               WIDTH: in FIELD: = DEFAULT_WIDTH;
               BASE: in NUMBER_BASE: = DEFAULT_BASE);
procedure PUT (ITEM: in NUM;
               WIDTH: in FIELD: = DEFAULT_WIDTH;
               BASE: in NUMBER_BASE: = DEFAULT_BASE);
```

Значение параметра ITEM выводят в виде целого литерала без подчеркиваний, порядка и ведущих нулей (если значение равно нулю, выводят один нуль), с предшествующим знаком минус, если значение отрицательное.

Если число символов выводимой последовательности меньше значения WIDTH, то она дополняется ведущими пробелами.

Если параметр BASE (заданный явно указанным значением или переменной DEFAULT_BASE) имеет значение десять, то числа выводятся по синтаксису десятичного литерала; в противном случае — по синтаксису литерала с основанием с использованием прописных букв.

```
procedure GET (FROM: in STRING; ITEM: out NUM; LAST: out POSITIVE);
```

Читает с начала строки, заданной параметром FROM, целое значение по тем же правилам, что и процедура GET, которая читает целое значение из файла; при этом конец строки рассматривается как признак конца файла. Через параметр ITEM возвращает значение типа NUM, соответствующее вве-

денной последовательности. Через параметр LAST возвращает значение индекса такое, что FROM (LAST) является последним читаемым символом.

Если введенная последовательность не соответствует правилам синтаксиса или если полученное значение не принадлежит подтипу NUM, то возбуждается исключение DATA_ERROR.

```
procedure PUT (TO: out STRING; ITEM: in NUM;
               BASE: in NUMBER_BASE: = DEFAULT_BASE);
```

Через параметр TO выводит значение параметра ITEM по тем же правилам, что и при выводе в файл; в качестве значения параметра WIDTH используется длина указанной строки.

Примеры:

```
package ЦЕЛ_ВВ is new INTEGER_IO (КОРОТКОЕ_ЦЕЛ);
use ЦЕЛ_ВВ;
- - в результате настройки получается формат по умолчанию:
- - DEFAULT_WIDTH = 4, DEFAULT_BASE = 10
PUT (126)                                  - - "b126"
PUT (–126,7)                               - - "bbb–126"
PUT (126, WIDTH => 13, BASE => 2);         - - "bbb2 # 1111110 #"
```

14.3.8. Ввод-вывод для вещественных типов

Следующие процедуры определены в настраиваемых пакетах FLOAT_IO и FIXED_IO, которые должны быть конкретизированы с соответствующими плавающим или фиксированным типом (указанным в спецификации параметром NUM).

Значения выводятся как десятичные литералы без подчеркиваний. Формат каждого выводимого значения состоит из поля FORE, десятичной точки, поля AFT, а также (при ненулевом значении параметра EXP) буквы E и поля EXP. Таким образом, возможны два формата:

FORE. AFT и FORE. AFT E EXP

без всяких пробелов между этими полями. Поле FORE может включать предшествующие пробелы и знак минус для отрицательных значений. Поле AFT состоит из одних лишь десятичных цифр (оно может оканчиваться нулями). Поле EXP состоит из знака (плюс или минус) и порядка (возможно, с предшествующими нулями).

Для плавающих типов длины этих полей по умолчанию определены описанными в пакете FLOAT_IO переменными:

```
DEFAULT_FORE: FIELD: = 2;
DEFAULT_AFT: FIELD: = NUM'DIGITS –1;
DEFAULT_EXP: FIELD: = 3;
```

Для фиксированных типов длины по умолчанию для этих полей задаются переменными, описанными в пакете FIXED_IO:

```
DEFAULT_FORE: FIELD: = NUM'FORE;
DEFAULT_AFT:FIELD: = NUM'AFT;
DEFAULT_EXP:FIELD: = 0;
```

Определены следующие процедуры:

```
procedure GET (FILE: in FILE_TYPE; ITEM: out NUM;
               WIDTH: in FIELD: = 0);
procedure GET (ITEM: out NUM; WIDTH: in FIELD: = 0);
```

При нулевом значении параметра WIDTH опускают все ведущие пропуски, признак конца строчки или признак конца страницы, читают знак плюс

или минус (если он есть); затем производят чтение в соответствии с синтаксисом вещественного литерала (он может быть литералом с основанием). При ненулевом значении параметра WIDTH вводят или ровно WIDTH символов или, если раньше встретился признак конца строчки, то вводят лишь символы, читаемые до этого признака конца (возможно, ни одного); в это количество включаются и все предшествующие пропуски.

Через параметр ITEM возвращают значение типа NUM, соответствующее введенной последовательности.

Если введенная последовательность не удовлетворяет требованиям синтаксиса или если полученное значение не принадлежит подтипу NUM, то возбуждается исключение DATA_ERROR.

```
procedure PUT (FILE: in FILE_TYPE; ITEM: in NUM;
               FORE: in FIELD: = DEFAULT_FORE;
               AFT: in FIELD: = DEFAULT_AFT;
               EXP: in FIELD: = DEFAULT_EXP);
procedure PUT (ITEM: in NUM;
               FORE: in FIELD: = DEFAULT_FORE;
               AFT: in FIELD: = DEFAULT_AFT;
               EXP: in FIELD: = DEFAULT_EXP);
```

Выводят значение параметра ITEM в виде десятичного литерала в формате, определяемом параметрами FORE, AFT и EXP. Если значение ITEM отрицательное, то в целую часть включен знак минус. При нулевом значении параметра EXP целая часть представляется таким количеством цифр, которое требуется для представления целой части значения ITEM. При необходимости значение FORE игнорируется. Если в значении ITEM нет целой части, то целая часть представляется цифрой 0.

Если значение EXP больше нуля, то целая часть представляется одной цифрой, отличной от нуля, кроме значения ITEM, равного 0.0.

В обоих случаях, если целая часть, включая знак минус, содержит менее FORE символов, она дополняется до этого количества предшествующими нулями. Дробная часть состоит из AFT цифр или, при AFT равном нулю, из одной цифры. Значение округляется; остаток, равный половине последнего разряда, может быть округлен как с избытком, так и с недостатком.

При нулевом значении EXP число выводится без порядка. Если EXP больше нуля, то при выводе порядка выводится столько цифр, сколько необходимо для представления порядка значения ITEM (для представления целой части этого значения используется один символ); первым символом является знак (плюс или минус). Если для представления порядка, включая знак, используется менее EXP символов, то это представление дополняется до требуемого количества символов предшествующими нулями. Для значения ITEM, равного 0.0, порядок равен нулю.

```
procedure GET (FROM: in STRING; ITEM: out NUM ; LAST: out POSITIVE);
```

Читает с начала строки, заданной параметром FROM, вещественное значение по тем же правилам, что и процедура GET, читающая вещественное значение из файла; при этом конец строки рассматривается как признак конца файла. Через параметр ITEM возвращает значение типа NUM, соответствующее введенной последовательности. Через параметр LAST возвращает

значение индекса такое, что FROM (LAST) является последним читаемым символом.

Если введенная последовательность не соответствует правилам синтаксиса или если полученное значение не принадлежит подтипу NUM, то возбуждается исключение DATA_ERROR.

```
procedure PUT (TO: out STRING; ITEM: in NUM;
                  AFT: in FIELD: = DEFAULT_AFT;
                 EXP: in FIELD: = DEFAULT_EXP);
```

Через параметр TO выводит значение параметра ITEM по тем же правилам, что и при выводе в файл; при этом в качестве FORE использует такое значение, чтобы общее число выводимых символов, включая предшествующие пробелы, соответствовало длине строки параметра TO.

Примеры:

```
package ВЕЩЕСТВ_ВВ is new FLOAT_IO (ВЕЩЕСТВ);
use ВЕЩЕСТВ_ВВ;
--в результате конкретизации получается формат по умолчанию; DEFAULT_EXP=3
X: ВЕЩЕСТВ: = –123.4567 -- digits 8 (см. 3.5.7)
PUT(X) -- формат по умолчанию "–1.2345670E+02"
PUT(X, FORE => 5, AFT => 3, EXP => 2); --"bbb–1.235E+2"
PUT(X, 5, 3, 0); --"b–123.457"
```

Примечание. Если положительное число, выводимое процедурой PUT в строку, заполняет строку целиком, без использования ведущих пробелов, то при выводе такого же отрицательного числа будет возбуждено исключение LAYOUT_ERROR.

14.3.9. Ввод-вывод для перечислимых типов

Описанные ниже процедуры определены в настраиваемом пакете ENUMERATION_IO, который должен быть конкретизирован с соответствующим перечислимым типом (указанным в спецификации параметром настройки ENUM).

При выводе значений для представления идентификаторов используются либо строчные, либо прописные буквы. Это задается параметром SET, который принадлежит перечислимому типу TYPE_SET:

```
type TYPE_SET is (LOWER_CASE, UPPER_CASE);
```

Формат (в который включаются и заключительные пробелы) может быть задан необязательным параметром ширины поля. Ширина поля по умолчанию и представление букв задаются описанными в настраиваемом пакете ENUMERATION_IO переменными:

```
DEFAULT_WIDTH: FIELD: = 0;
DEFAULT_SETTING: TYPE_SET: = UPPER_CASE;
```

Определены следующие процедуры:

```
procedure GET (FILE: in FILE_TYPE; ITEM: out ENUM);
procedure GET (ITEM: out ENUM);
```

После игнорирования предшествующих пропусков, признака конца строчки и признака конца страницы читают или идентификатор (строчные и прописные буквы считаются эквивалентными), или символьный литерал (включая апострофы); чтение производится в соответствии с синтаксисом соответствующих лексем. Через параметр ITEM возвращают значение, соответствующее введенной последовательности.

Если введенная последовательность не удовлетворяет правилам синтаксиса или если идентификатор или символьный литерал не соответствуют никакому значению подтипа ENUM, то возбуждается исключение DATA_ERROR.

```
procedure PUT (FILE: in FILE_TYPE;
               ITEM: in ENUM;
               WIDTH: in FIELD: = DEFAULT_WIDTH;
               SET: in TYPE_SET: = DEFAULT_SETTING);
procedure PUT (ITEM: in ENUM;
               WIDTH: in FIELD: = DEFAULT_WIDTH;
               SET: in TYPE_SET: = DEFAULT_SETTING);
```

Выводят значение параметра ITEM как литерал перечисления (либо идентификатор, либо символьный литерал). Необязательный параметр SET указывает, какие буквы – строчные или прописные – следует использовать для представления идентификаторов (для символьных литералов он игнорируется). Если число выводимых символом меньше значения параметра WIDTH, то после них выводят пробелы, дополняющие число символов до WIDTH.

```
procedure GET (FROM: in STRING;
               ITEM: out ENUM;
               LAST: out POSITIVE);
```

Читает с начала строки, заданной параметром FROM, значение перечислимого типа по тем же правилам, что и процедура GET, читающая значение перечислимого типа из файла; при этом конец строки рассматривает как признак конца файла. Через параметр ITEM возвращает значение типа ENUM, соответствующее введенной последовательности. Через параметр LAST возвращает значение индекса такое, что FROM (LAST) является последним читаемым символом.

Если введенная последовательность не соответствует правилам синтаксиса или если идентификатор или символьный литерал не соответствуют никакому значению подтипа ENUM, то возбуждается исключение DATA_ERROR.

```
procedure PUT (TO: out STRING;
               ITEM: in ENUM;
               SET: in TYPE_SET: = DEFAULT_SETTING);
```

Выводит в строку, заданную параметром TO, значение параметра ITEM по тем же правилам, что и при выводе в файл; в качестве значения параметра WIDTH используется длина указанной строки.

Хотя спецификация пакета ENUMERATION_IO допускает конкретизацию с соответствующим целым типом, это не является целью данного пакета; язык не определяет результата такой конкретизации.

Примечание. Процедуры PUT для символов и перечислимых значений имеют определенные различия:

```
TEXT_IO.PUT('A'); - - выводит символ A
package СИМВ_ВВ is new TEXT_IO.ENUMERATION_IO (CHARACTER);
СИМВ_ВВ. PUT('A'); - - выводит символ 'A',
                   - - заключенный в одинарные
                   - - кавычки
```

Тип BOOLEAN является перечислимым типом, поэтому пакет ENUMERATION_IO может быть настроен на этот тип.

14.3.10. Спецификация пакета ввода-вывода текста

```
with IO_EXCEPTIONS;
package TEXT_IO is
  type FILE_TYPE is limited private;
  type FILE_MODE is (IN_FILE, OUT_FILE);
  type COUNT is range 0. определяется_реализацией;
  subtype POSITIVE_COUNT is COUNT range 1. . COUNT'LAST;
     UNBOUNDED: constant COUNT: = 0; - - длина строчки и страницы
    subtype FIELD is INTEGER range 0. определяется _реализацией;
    subtype NUMBER_BASE is INTEGER range 2. .16;
    type TYPE_SET is (LOWER_CASE, UPPER_CASE);
    - - управление файлами
    procedure CREATE (FILE: in out FILE_TYPE;
                      MODE: in FILE_MODE: = OUT_FILE;
                      NAME: in STRING: = " ";
                      FORM: in STRING: = " ");
    procedure OPEN (FILE: in out FILE_TYPE;
                    MODE: in FILE_MODE;
                    NAME: in STRING;
                    FORM: in STRING: = " ");
    procedure CLOSE (FILE: in out FILE_TYPE);
    procedure DELETE (FILE: in out FILE_TYPE);
    procedure RESET (FILE: in out FILE_TYPE,MODE: in FILE_MODE);
    procedure RESET (FILE: in out FILE_TYPE);
    function MODE (FILE: in FILE_TYPE( return FILE_MODE;
    function NAME (FILE: in FILE_TYPE) return STRING;
    function FORM (FILE: in FILE_TYPE) return STRING;
    function IS_OPEN (FILE: in FILE_TYPE) return BOOLEAN;
    - - управление файлами ввода и вывода по умолчанию
    procedure SET_INPUT (FILE: in FILE_TYPE);
    procedure SET_OUTPUT (FILE: in FILE_TYPE);
    function STANDARD_INPUT return FILE_TYPE;
    function STANDARD_OUTPUT return FILE_TYPE;
    function CURRENT_INPUT return FILE_TYPE;
    function CURRENT_OUTPUT return FILE_ TYPE ;
    - - спецификация длин строчки и страницы
    procedure SET_LINE_LENGTH (FILE: in FILE_TYPE; TO: in COUNT);
    procedure SET_LINE_LENGTH (TO: in COUNT);
    procedure SET_PAGE_LENGTH (FILE: in FILE_TYPE; TO: in COUNT);
    procedure SET_PAGE_LENGTH (TO: in COUNT);
    function LINE_LENGTH (FILE: in FILE_TYPE) return COUNT;
    function LINE_LENGTH return COUNT;
    function PAGE_LENGTH (FILE: in FILE_TYPE) return COUNT;
    function PAGE_LENGTH return  COUNT;
    - - управление колонкой, строчкой и страницей
    procedure NEW_LINE (FILE: in FILE_TYPE;
                        SPACING: in POSITIVE_COUNT: = 1);
    procedure NEW_LINE (SPACING: in POSITIVE_COUNT: = 1);
    procedure SKIP_LINE (FILE: in FILE_TYPE;
                        SPACING: in POSITIVE_COUNT: = 1);
    procedure SKIP_LINE (SPACING: in POSITIVE_COUNT: = 1);
    function END_OF_LINE (FILE: in FILE_TYPE) return BOOLEAN·
    function END_OF_LINE return BOOLEAN;
    procedure NEW_PAGE (FILE: in FILE_TYPE);
    procedure NEW_PAGE;
```

```
procedure SKIP_PAGE (FILE: in FILE_TYPE);
procedure SKIP_PAGE;
function END_OF_PAGE (FILE: in FILE_TYPE) return BOOLEAN;
function END_OF_PAGE return BOOLEAN;
function END_OF_FILE (FILE: in FILE_TYPE) return BOOLEAN;
function END_OF_FILE return BOOLEAN;
procedure SET_COL (FILE: in FILE_TYPE; TO: in POSITIVE_COUNT);
procedure SET_COL (TO: in POSITIVE_COUNT);
procedure SET_LINE (FILE: in FILE_TYPE; TO: in POSITIVE_COUNT);
procedure SET_LINE (TO: in POSITIVE_COUNT);
function COL (FILE: in FILE_TYPE) return POSITIVE_COUNT;
function COL return POSITIVE_COUNT;
function LINE (FILE: in FILE_TYPE) return POSITIVE_COUNT;
function LINE return POSITIVE_COUNT;
function PAGE (FILE: in FILE_TYPE) return POSITIVE_COUNT;
function PAGE return POSITIVE_COUNT;
-- символьный ввод-вывод
procedure GET (FILE: in FILE_TYPE; ITEM: out CHARACTER);
procedure GET (ITEM: out CHARACTER);
procedure PUT (FILE: in FILE_TYPE; ITEM: in CHARACTER);
procedure PUT (ITEM: in CHARACTER);
-- строковый ввод-вывод
procedure GET (FILE: in FILE_TYPE; ITEM: out STRING);
procedure GET (ITEM: out STRING);
procedure PUT (FILE: in FILE_TYPE; ITEM: in STRING);
procedure PUT (ITEM: in STRING);
procedure GET_LINE (FILE: in FILE_TYPE;
                    ITEM: out STRING;
                    LAST: out NATURAL);
procedure GET_LINE (ITEM: out STRING; LAST: out NATURAL);
procedure PUT_LINE (FILE: in FILE_TYPE; ITEM: in STRING);
procedure PUT_LINE (ITEM: in STRING);
-- настраиваемый пакет для ввода-вывода целых типов
generic
  type NUM is range < >;
package INTEGER_IO is
    DEFAULT_WIDTH: FIELD: = NUM'WIDTH;
    DEFAULT_BASE: NUMBER_BASE: = 10;
  procedure GET (FILE: in FILE_TYPE; ITEM: out NUM; WIDTH: in FIELD: = 0);
  procedure GET (ITEM: out NUM; WIDTH: in FIELD: = 0);
  procedure PUT (FILE: in FILE_TYPE;
                 ITEM: in NUM;
                 WIDTH: in FIELD:   = DEFAULT_WIDTH;
                 BASE: in NUMBER_BASE: = DEFAULT_BASE);
  procedure PUT (ITEM: in NUM;
                 WIDTH: in FIELD: = DEFAULT_WIDTH;
                 BASE: in NUMBER_BASE: = DEFAULT_BASE);
  procedure GET (FROM: in STRING; ITEM: out NUM; LAST: out POSITIVE);
  procedure PUT (TO: out STRING;
                 ITEM: in NUM;
                 BASE: in NUMBER_BASE: = DEFAULT_BASE);
end INTEGER_IO;
-- настраиваемый пакет для ввода-вывода вещественных типов
generic
  type NUM is digits < >;
```

```
package FLOAT_IO is
  DEFAULT_FORE: FIELD: = 2;
  DEFAULT_AFT: FIELD: = NUM'DIGITS –1;
  DEFAULT_EXP: FIELD: = 3;
  procedure GET (FILE: in FILE_TYPE; ITEM: out NUM; WIDTH: in FIELD: = 0);
  procedure GET (ITEM: out NUM; WIDTH: in FIELD: = 0) ;
  procedure PUT (FILE: in FILE_TYPE;
                 ITEM: in NUM;
                 FORE: in FIELD: = DEFAULT_FORE;
                 AFT: in FIELD: = DEFAULT_AFT;
                 EXP: in FIELD: = DEFAULT_EXP);
  procedure PUT (ITEM: in NUM;
                 FORE: in FIELD: = DEFAULT_FORE;
                 AFT: in FIELD: = DEFAULT_AFT;
                 EXP: FIELD: = DEFAULT_EXP);
  procedure GET (FROM: in STRING; ITEM: out NUM; LAST: out POSITIVE);
  procedure PUT (TO: out STRING;
                 ITEM: in NUM;
                 AFT: in FIELD: = DEFAULT_AFT;
                 EXP: in FIELD: = DEFAULT_EXP);
end FLOAT_IO;
generic
  type NUM is delta < >;
package FIXED_IO is
  DEFAULT_FORE: FIELD: = NUM'FORE;
  DEFAULT_AFT: FIELD: = NUM'AFT;
  DEFAULT_EXP: FIELD: = 0;
  procedure GET (FILE: in FILE_TYPE; ITEM: out NUM; WIDTH: in FIELD: = 0);
  procedure GET (ITEM: out NUM; WIDTH: in FIELD: = 0);
  procedure PUT (FILE: in FILE_TYPE;
                 ITEM: in NUM;
                 FORE: in FIELD. = DEFAULT_FORE;
                 AFT: in FIELD: = DEFAULT_AFT;
                 EXP: in FIELD: = DEFAULT_EXP);
  procedure PUT (ITEM: in NUM;
                 FORE: in FIELD: = DEFAULT_FORE;
                 AFT: in FIELD: = DEFAULT_AFT;
                 EXP: in FIELD: = DEFAULT_EXP);
  procedure GET (FROM: in STRING; ITEM: out NUM; LAST: out POSITIVE);
  procedure PUT (TO: out STRING; ITEM: in NUM;
                 AFT: in FIELD: = DEFAULT_AFT;
                 EXP: in FIELD: = DEFAULT_EXP);
end FIXED_IO;
- - настраиваемый пакет для ввода-вывода перечислимых типов
generic
  type ENUM is (< >);
package ENUMERATION_IO is
  DEFAULT_WIDTH: FIELD: = 0;
  DEFAULT_SETTING: TYPE_SET: = UPPER_CASE;
  procedure GET (FILE: in FILE_TYPE; ITEM: out ENUM);
  procedure GET (ITEM: out ENUM);
  procedure PUT (FILE: in FILE_TYPE; ITEM: in ENUM;
                 WIDTH: in FIELD: = DEFAULT_WIDTH;
                 SET: in TYPE_SET: = DEFAULT_SETTING);
```

```
    procedure PUT (ITEM: in ENUM;
                   WIDTH: in FIELD: = DEFAULT_WIDTH;
                   SET: in TYPE_SET: = DEFAULT_SETTING);
    procedure GET (FROM: in STRING; ITEM: out ENUM;
                   LAST: out POSITIVE);
    procedure PUT (TO: out STRING, ITEM: in ENUM;
                   SET: in TYPE_SET: = DEFAULT_SETTING);
  end ENUMERATION_IO;
    - - исключения
    STATUS_ERROR: exception renames IO_EXCEPTIONS.STATUS_ERROR,
    MODE_ERROR: exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR: exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR: exception renames IO_EXCEPTIONS.USE_ERROR;
    DEVICE_ERROR: exception renames IO_EXCEPTIONS.DEVICE_ERROR;
    END_ERROR: exception renames IO_EXCEPTIONS.END_ERROR;
    DATA_ERROR: exception renames IO_EXCEPTIONS.DATA_ERROR;
    LAYOUT_ERROR: exception renames IO_EXCEPTIONS.LAYOUT_ERROR;
  private
    – определяется реализацией
  end TEXT_IO,
```

14.4. **Исключения при вводе-выводе**

В этом разделе описаны исключения, которые могут быть возбуждены при выполнении операций ввода-вывода. Они описаны в пакете IO_EXCEPTIONS; определение этого пакета приведено в разд. 14.5. Этот пакет упоминается в спецификаторах контекста каждого из трех описанных пакетов ввода-вывода. Для исключений NAME_ERROR, USE_ERROR и DEVICE_ERROR описаны лишь общие характеристики условий возбуждения этих исключений; детальное описание должно быть приведено в руководстве по реализации в соответствии с обязательным приложением 4. Если существует более чем одно ошибочное условие, соответствующее одному исключению, то возбуждается то из них, которое раньше описано в данном ниже списке исключений.

Исключение STATUS_ERROR возбуждается при попытке выполнить действия над еще не открытым файлом или при попытке открыть уже открытый файл.

Исключение MODE_ERROR возбуждается при попытке чтения или проверке конца файла с текущим видом OUT_FILE, а также при попытке записи в файл с текущим видом IN_FILE. Для пакета TEXT_IO исключение MODE_ERROR возбуждается также при использовании файла с текущим видом OUT_FILE в качестве параметра подпрограмм SET_INPUT, SKIP_LINE, END_OF_LINE, SKIP_PAGE и END_OF_PAGE и при использовании файла с текущим видом IN_FILE в качестве параметра подпрограмм SET_OUTPUT, SET_LINE_LENGTH, SET_PAGE_LENGTH, LINE_LENGTH, PAGE_LENGTH, NEW_LINE или NEW_PAGE.

Исключение NAME_ERROR возбуждается при вызове процедур CREATE и OPEN, если строка, заданная параметром NAME, не позволяет идентифицировать внешний файл. Например, это исключение возбуждается, если такая строка вообще недопустима или если ей либо не соответствует ни один внешний файл, либо соответствует несколько внешних файлов.

Исключение USE_ERROR возбуждается при попытке выполнить операцию, не разрешенную по причинам, зависящим от характеристик внешнего файла. Например, процедурой CREATE такое исключение может быть возбуждено при задании параметром MODE вида OUT_FILE, а параметром FROM устройства, допускающего только ввод, либо при задании параметром FORM неверных прав доступа, либо, если внешний файл с данным именем уже существует и перезапись недопустима.

Исключение DEVICE_ERROR возбуждается при невозможности завершения операции ввода-вывода из-за неисправностей в используемой системе.

Исключение END_ERROR возбуждается при попытке пропустить признак конца файла (чтение за концом файла).

Исключение DATA_ERROR может быть возбуждено процедурой READ, если читаемый элемент нельзя интерпретировать как значение требуемого типа. Это же исключение возбуждается процедурой GET (определенной в пакете TEXT_IO), если введенная последовательность символов не соответствует требуемому синтаксису или если введенное значение не принадлежит диапазону требуемого типа или подтипа.

Исключение LAYOUT_ERROR возбуждается (в текстовом вводе-выводе) при вызове функции COL, LINE или PAGE, если возвращаемое значение превышает COUNT'LAST. Это же исключение возбуждается при выводе: при попытке установить номер столбца или строчки, превышающий заданную максимальную длину строчки или страницы (кроме случая неограниченной длины строчки или страницы). Оно также возбуждается процедурой PUT при попытке вывести в строку слишком большое количество символов.

14.5. **Спецификация пакета исключений ввода-вывода**

В этом пакете определены исключения, необходимые для работы пакетов SEQUENTIAL_IO, DIRECT_IO и TEXT_IO.

```
package IO_EXCEPTIONS is
  STATUS_ERROR: exception;
  MODE_ERROR: exception;
  NAME_ERROR: exception;
  USE_ERROR: exception;
  DEVICE_ERROR: exception;
  END_ERROR: exception;
  DATA_ERROR: exception;
  LAYOUT_ERROR: exception;
end IO_EXCEPTIONS;
```

14.6. **Ввод-вывод низкого уровня**

Операция ввода-вывода низкого уровня – это операция, выполняющаяся на физическом устройстве. Для выполнения таких операций используется одна из (совмещенных) предопределенных процедур SEND_CONTROL и RECEIVE_CONTROL.

Процедура SEND_CONTROL может быть использована для посылки управляющей информации физическому устройству. Процедура RECEIVE_CONTROL может быть использована для организации выполнения операции ввода-вывода, связанной с запросом информации от физического устройства.

Эти процедуры описаны в стандартном пакете LOW_LEVEL_IO. Каждая из них имеет два параметра, задающие устройство и данные. Однако сорт и формат управляющей информации зависит от физических характеристик машины и устройств; поэтому типы параметров зависят от реализации. Возможно определение совмещенных подпрограмм, управляющих различными устройствами.

Видимый раздел пакета, определяющего такие процедуры, имеет следующую структуру:

```
package LOW_LEVEL_IO is
  -- описания возможных типов параметров DEVICE и DATA
  -- описания совмещенных процедур над такими типами:
  procedure SEND_CONTROL (DEVICE: тип_устройства;
                          DATA: in out тип_данных);
  procedure RECEIVE_CONTROL (DEVICE: тип_устройства;
                             DATA: in out тип_данных);
end;
```

Тела процедур SEND_CONTROL и RECEIVE_CONTROL для различных устройств могут быть размещены в теле пакета LOW_LEVEL_IO. Эти тела процедур могут быть записаны операторами кода.

14.7. **Пример ввода-вывода**

В примере показано использование некоторых средств ввода-вывода текстов в режиме диалога с пользователем за терминалом. Пользователю предлагается выбрать цвет; программа в соответствии с описью выдает число предметов этого цвета на складе. Используются файлы ввода и вывода по умолчанию. Для простоты все необходимые конкретизации настройки заданы в одной подпрограмме; на практике для этого мог бы использоваться самостоятельный пакет.

```
with TEXT_IO; use TEXT_IO;
procedure ДИАЛОГ is
   type ЦВЕТ is (БЕЛЫЙ, КРАСНЫЙ, ОРАНЖЕВЫЙ, ЖЕЛТЫЙ, ЗЕЛЕНЫЙ,
                 СИНИЙ, КОРИЧНЕВЫЙ);
  package ВВ_ЦВЕТА is new ENUMERATION_IO (ENUM => ЦВЕТ);
  package ВВ_ЧИСЛА is new INTEGER_IO (INTEGER);
  use ВВ_ЦВЕТА, ВВ_ЧИСЛА;
  ОПИСЬ: array (ЦВЕТ) of INTEGER: = (20, 17, 43, 10, 28, 173, 87);
  ВАРИАНТ: ЦВЕТ;
  procedure ВЫБОР_ЦВЕТА (ВЫБОР: out ЦВЕТ) is
  begin
    loop
      begin
        PUT ("ВЫБРАННЫЙ ЦВЕТ: "); -- обращение к пользователю
        GET (ВЫБОР); -- вводит набранный цвет или возбуждает исключение
        return;
      exception
        when DATA_ERROR =>
          PUT ("ОШИБОЧНЫЙ ЦВЕТ, ПОВТОРИТЕ НАБОР");
          -- пользователь должен набрать новую строчку
          NEW_LINE (2);
          -- завершение выполнения оператора блока
      end;
```

```
  end loop, - - повторение оператора блока, пока не будет получен правильный цвет
end;
begin - - операторы процедуры ДИАЛОГ;
  ВВ_ЧИСЛА.DEFAULT_WIDTH: = 5;
  loop
     ВЫБОР_ЦВЕТА  (ВАРИАНТ);
     - - пользователь набирает цвет и начинает новую строчку
  SET_COL (5); PUT (ВАРИАНТ); PUT ("ВСЕГО ПРЕДМЕТОВ: ");
  SET_COL (40); PUT  (ОПИСЬ (ВАРИАНТ)); - - ширина по умолчанию равна 5
  NEW_LINE;
  end loop;
end ДИАЛОГ;
```

Пример диалога (набранное пользователем выделено курсивом):

```
ВЫБРАННЫЙ ЦВЕТ: ЧЕРНЫЙ
ОШИБОЧНЫЙ ЦВЕТ, ПОВТОРИТЕ НАБОР
ВЫБРАННЫЙ ЦВЕТ: СИНИЙ
                СИНИЙ ВСЕГО ПРЕДМЕТОВ: 173
ВЫБРАННЫЙ ЦВЕТ: ЖЕЛТЫЙ
                ЖЕЛТЫЙ ВСЕГО ПРЕДМЕТОВ: 10
```

ПРИЛОЖЕНИЕ 1
Обязательное

АТРИБУТЫ, ПРЕДОПРЕДЕЛЕННЫЕ В ЯЗЫКЕ

Ниже представлена сводка определений предопределенных атрибутов, которые рассматривались в описании языка. (В скобках указан пояснительный перевод.)

P'ADDRESS (P'АДРЕС). Префикс P обозначает объект, программный модуль, метку или вход. Вырабатывает адрес первого кванта памяти, отведенной под P. Для подпрограммы, пакета, задачи или метки это значение ссылается на машинный код, связанный с соответствующим телом или оператором. Для входа с заданным спецификатором адреса это значение ссылается на соответствующее аппаратное прерывание. Значение этого атрибута имеет предопределенный в пакете SYSTEM тип ADDRESS (см. разд. 13.7.2).

P'AFT (P'ЦИФР_ДРОБИ). Префикс P обозначает фиксированный подтип. Вырабатывает число десятичных цифр после точки, необходимых для обеспечения точности подтипа P, если только атрибут DELTA подтипа P не превышает значения 0.1, для которого атрибут вырабатывает значение единицы. (P'AFT – это наименьшее положительное число K, для которого (10 ** K) * P'DELTA больше или равно 1). Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 3.5.10).

P'BASE (P'БАЗОВЫЙ). Префикс P обозначает тип или подтип. Этот атрибут обозначает базовый тип P и может быть только префиксом имени другого атрибута, например, P'BASE'FIRST (см. разд. 3.3.3).

P'CALLABLE (P'ВЫЗЫВАЕМА). Префикс P – это объект задачного типа. Вырабатывает значение FALSE, когда выполнение задачи P либо закончено, либо завершено, либо задача находится в аварийном состоянии. В остальных случаях вырабатывает значение TRUE. Значение этого атрибута имеет предопределенный тип BOOLEAN (см. разд. 9.9).

P'CONSTRAINED (P'ОГРАНИЧЕН). Префикс P обозначает объект некоторого типа с дискриминантами. Вырабатывает значение TRUE, если ограничение дискриминантов наложено на объект P или если объект – константа (включая формальный параметр или формальный параметр настройки вида in). В остальных случаях вырабатывает значение FALSE. Если P формальный параметр настройки вида **in out** или если P – формальный параметр вида **in out** или **out**, а обозначение типа, заданное в соответствующей спецификации параметра, обозначает неограниченный тип с дискриминантами, то значение этого атрибута получается из значения атрибута соответствующего фактического параметра. Значение атрибута имеет предопределенный тип BOOLEAN (см. разд. 3.7.4).

P'CONSTRAINED (P'ОГРАНИЧЕН). Префикс P обозначает личный тип или подтип. Вырабатывает значение FALSE, если P обозначает неограниченный неформальный личный тип с дискриминантами, вырабатывает это же значение, если P обозначает настраиваемый формальный личный тип, а соответствующий фактический подтип – это либо неограниченный тип с дискриминантами, либо неограниченный индексируемый тип. В остальных случаях вырабатывает значение TRUE. Значение этого атрибута имеет предопределенный тип BOOLEAN (см. разд. 7.4.2).

P'COUNT (P'ЧИСЛО_ВЫЗОВОВ). Префикс P обозначает вход задачи. Вырабатывает число вызовов входа, присутствующих в очереди этого входа (если атрибут вычисляется внутри оператора принятия входа P, то в это число не входит вызывающая задача). Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 9.9).

P'DELTA (P'ДЕЛЬТА). Префикс P обозначает фиксированный подтип. Вырабатывает значение дельты, заданной определением точности фиксированного типа для подтипа P. Значение этого атрибута имеет *универсальный_вещественный* тип (см. разд. 3.5.10).

P'DIGITS (Р'ЦИФР). Префикс Р обозначает плавающий подтип. Вырабатывает число десятичных цифр в десятичной мантиссе модельных чисел подтипа Р. (Этот атрибут вырабатывает число Д, определенное в разд. 3.5.7). Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 3.5.8).

P'EMAX (Р'МАКС_ПОРЯДОК). Префикс Р обозначает плавающий подтип. Вырабатывает наибольшее значение порядка двоичной канонической формы модельных чисел подтипа Р. (Этот атрибут вырабатывает произведение 4 * В, определенное в разд. 3.5.7) Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 3.5.8).

P'EPSILON (Р'ЭПСИЛОН). Префикс Р обозначает плавающий подтип. Вырабатывает абсолютное значение разности между модельным числом 1.0 и следующим модельным числом подтипа Р. Значение этого атрибута имеет *универсальный_вещественный* тип (см. разд. 3.5.8).

P'FIRST (Р'ПЕРВЫЙ). Префикс Р обозначает скалярный тип или подтип скалярного типа. Вырабатывает значение нижней границы Р. Значение этого атрибута имеет тип Р (см. разд. 3.5).

P'FIRST. Префикс Р соответствует индексируемому типу или обозначает ограниченный индексируемый подтип. Вырабатывает значение нижней границы диапазона первого индекса. Значение этого атрибута имеет тот же тип, что и тип значения нижней границы (см. разд. 3.6.2 и 3.8.2).

P'FIRST(K). Префикс Р соответствует индексируемому типу или обозначает ограниченный индексируемый подтип. Вырабатывает значение нижней границы диапазона К-го индекса. Значение этого атрибута имеет тот же тип, что и указанная нижняя граница. Аргумент К должен быть статическим выражением типа *универсальный_целый*. Значение К должно быть положительным (ненулевым) и не превосходить размерности массива (см. разд. 3.6.2 и 3.8.2).

P'FIRST_BIT. Префикс Р обозначает компонент записи. Вырабатывает величину смещения первого бита относительно начала первого кванта памяти, занимаемой этим компонентом. Величина смещения измеряется числом битов. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.2).

P'FORE (Р'ЦИФР_ИЗОБРАЖЕНИЯ). Префикс Р обозначает фиксированный подтип. Вырабатывает минимальное число символов, необходимых для десятичного представления целой части любого значения подтипа Р в предположении, что это представление не включает порядок, но включает односимвольный префикс, который является либо знаком минус, либо пробелом. (Это минимальное число не учитывает предшествующие нули и символы подчеркивания и, по меньшей мере, равно двум.) Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 3.5.10).

P'IMAGE (Р'ОБРАЗ). Префикс Р обозначает дискретный тип или подтип. Этот атрибут представляет собой функцию с одним параметром. Фактический параметр Х должен быть значением базового тип Р. Тип результата – предопределенный тип STRING. Результат представляет собой образ значения Х, т. е. последовательность символов, представляющих изображение значения. Образу целого значения соответствует десятичный литерал без символов подчеркивания, предшествующих нулей, порядка и последующих пробелов, но с одним символом слева, который представляет собой либо минус, либо пробел. Образом литерала перечисления является либо соответствующий идентификатор из прописных букв, либо соответствующий символьный литерал (включая два апострофа) без предшествующих и последующих пробелов. Образ символа, отличного от графического, определяется реализацией (см. разд. 3.5.5).

P'LARGE (Р'НАИБОЛЬШИЙ). Префикс Р обозначает вещественный подтип. Вырабатывает наибольшее положительное модельное число подтипа Р. Значение этого атрибута имеет *универсальный_вещественный* тип (см. разд. 3.5.8 и 3.5.10).

P'LAST (Р'ПОСЛЕДНИЙ). Префикс Р обозначает скалярный тип или подтип скалярного типа. Вырабатывает значение верхней границы Р. Значение этого атрибута имеет тип Р (см. разд. 3.5).

P'LAST. Префикс P соответствует индексируемому типу или обозначает ограниченный индексируемый подтип. Вырабатывает значение верхней границы диапазона первого индекса. Значение атрибута имеет тип верхней границы (см. разд. 3.6.2 и 3.8.2).

P'LAST(K). Префикс P соответствует индексируемому типу или обозначает ограниченный индексируемый подтип. Вырабатывает значение верхней границы диапазона K-го индекса. Значение этого атрибута имеет тот же тип, что и верхняя граница. Аргумент K должен быть статическим выражением типа *универсальный_целый*. Значение K должно быть положительным (ненулевым) и не превышать размерности массива (см. разд. 3.6.2 и 3.8.2).

P'LAST_BIT (P'ПОСЛЕДНИЙ БИТ). Префикс P обозначает компонент записи. Вырабатывает величину смещения последнего бита относительно первого кванта памяти, занимаемой этим компонентом. Величина смещения измеряется числом битов. Значение атрибута имеет *универсальный_целый* тип (см. разд. 13.7.2).

P'LENGTH (P'КОЛИЧЕСТВО_ЗНАЧЕНИЙ). Префикс P соответствует индексируемому типу или обозначает ограниченный индексируемый подтип. Вырабатывает число значений диапазона первого индекса (нуль для пустого диапазона). Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 3.6.2).

P'LENGTH(K). Префикс P соответствует индексируемому типу или обозначает ограниченный индексируемый подтипа. Вырабатывает число значений в диапазоне K-го индекса (нуль для пустого диапазона). Значение этого атрибута имеет *универсальный_целый* тип. Аргумент K должен быть статическим выражением типа *универсальный_целый*. Значение K должно быть положительным (ненулевым) и не должно превышать размерности массива (см. разд. 3.6.2 и 3.8.2).

P'MACHINE_EMAX (P'МАКС_ПОРЯДОК_В_ЭВМ). Префикс P обозначает плавающий тип или подтип. Вырабатывает наибольшее значение порядка машинного представления базового типа P. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.3).

P'MACHINE_EMIN (P'МИН_ПОРЯДОК_В_ЭВМ). Префикс P обозначает плавающий тип или подтип. Вырабатывает наименьшее (наибольшее по модулю отрицательное) значение *порядка* машинного представления базового типа P. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.3).

P'MACHINE_MANTISSA (P'МАНТИССА_В_ЭВМ). Префикс P обозначает плавающий тип или подтип. Вырабатывает число цифр в *мантиссе* машинного представления базового типа P (цифры являются расширенными цифрами из диапазона 0 . . P'MACHINE_RADIX–1). Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.3).

P'MACHINE_OVERFLOWS (P'ПЕРЕПОЛНЕНИЕ_В_ЭВМ). Префикс P обозначает вещественный тип или подтип. Вырабатывает значение TRUE, если каждая предопределенная операция над значениями базового типа P либо возвращает точный результат, либо возбуждает исключение NUMERIC_ERROR при переполнении. В противном случае вырабатывает значение FALSE. Значение этого атрибута имеет предопределенный тип BOOLEAN (см. разд. 13.7.3).

P'MACHINE_RADIX (P'ОСНОВАНИЕ_В_ЭВМ). Префикс P обозначает плавающий тип или подтип. Вырабатывает значение *основания*, используемое в машинном представлении базового типа P. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.3).

P'MACHINE_ROUNDS (P'ОКРУГЛЕНИЕ_В_ЭВМ). Префикс P обозначает вещественный тип или подтип. Вырабатывает значение TRUE, если каждая предопределенная арифметическая операция над значениями базового типа P либо возвращает точный результат, либо осуществляет округление. В противном случае вырабатывает значение FALSE. Значение этого атрибута имеет предопределенный тип BOOLEAN (см. разд. 13.7.3).

P'MANTISSA (P'МАНТИССА). Префикс P обозначает вещественный подтип. Вырабатывает число двоичных цифр мантиссы модельных чисел подтипа P. (Этот

атрибут вырабатывает число B, введенное в разд. 3.5.7 для плавающего типа и в разд. 3.5.9 для фиксированного типа.) Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 3.5.8 и 3.5.10).

P'POS (Р'ПОЗИЦ). Префикс P обозначает дискретный тип или подтип. Этот атрибут является функцией с одним параметром. Фактический параметр X должен быть значением базового типа P. Тип результата – *универсальный_целый*. Результатом является порядковый номер позиции для значения фактического параметра (см. разд. 3.5.5).

P'POSITION (Р'ПОЗИЦИЯ). Префикс P обозначает компонент записи. Вырабатывает величину смещения первого кванта памяти, занятого этим компонентом, относительно первого кванта памяти, занимаемой записью. Величина смещения измеряется числом квантов. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.2).

P'PRED (Р'ПРЕДШ). Префикс P обозначает дискретный тип или подтип. Этот атрибут является функцией с одним параметром. Фактический параметр X должен быть значением базового типа P. Тип результата – базовый тип P. Результатом является значение с номером позиции на единицу меньшим номера позиции для значения X. Если X равен P'BASE'FIRST, то возбуждается исключение CONSTRAINT_ERROR (см. разд. 3.5.5).

P'RANGE (Р'ДИАПАЗОН). Префикс P соответствует индексируемому типу или обозначает ограниченный индексируемый подтип. Вырабатывает диапазон первого индекса P, т. е. диапазон P'FIRST . . P'LAST (см. разд. 3.6.2).

P'RANGE(K). Префикс P соответствует индексируемому типу или обозначает ограниченный индексируемый подтип. Вырабатывает диапазон *K*-го индекса *P*, т. е. диапазон P'FIRST(K) . . P'LAST(K) (см. разд. 3.6.2).

P'SAFE_EMAX (Р'ХРАНИМЫЙ_МАКС_ПОР). Префикс P обозначает плавающий тип или подтип. Вырабатывает наибольшее значение порядка двоичной канонической формы хранимых чисел базового типа P. (Этот атрибут вырабатывает число E, определенное в разд. 3.5.7.) Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 3.5.8).

P'SAFA_LARGE (Р'НАИБОЛЬШИЙ_ХРАНИМЫЙ). Префикс P обозначает вещественный тип или подтип. Вырабатывает наибольшее положительное хранимое число базового типа P. Значение этого атрибута имеет *универсальный_вещественный* тип (см. разд. 3.5.8 и 3.5.10).

P'SAFE_SMALL (Р'ХРАНИМЫЙ_ДИСКРЕТ). Префикс P обозначает вещественный тип или подтип. Вырабатывает наименьшее положительное (ненулевое) хранимое число базового типа P. Значение этого атрибута имеет *универсальный_вещественный* тип (см. разд. 3.5.8 и 3.5.10).

P'SIZE (Р'РАЗМЕР). Префикс P обозначает тип или подтип. Вырабатывает минимальное число битов, необходимое реализации для представления любого возможного значения объекта типа или подтипа P. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.2).

P'SIZE. Префикс P обозначает объект. Вырабатывает число битов, отведенных для размещения объекта. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.2).

P'SMALL (Р'ДИСКРЕТ). Префикс P обозначает вещественный подтип. Вырабатывает наименьшее положительное (ненулевое) модельное число подтипа P. Значение этого атрибута имеет *универсальный_вещественный* тип (см. разд. 3.5.8 и 3.5.10).

P'STORAGE_SIZE (Р'РАЗМЕР_ПАМЯТИ). Префикс P обозначает ссылочный тип или подтип. Вырабатывает общее число квантов памяти, резервируемых для соответствующего базовому типа P набора. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.2).

P'STORAGE_SIZE. Префикс обозначает задачный тип или задачу. Вырабатывает общее число квантов памяти, резервируемых для каждой активизации задачи типа P или объекта P задачного типа. Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 13.7.2).

P'SUCC (P'СЛЕД). Префикс P обозначает дискретный тип или подтип. Этот атрибут является функцией с одним параметром. Фактический параметр X должен быть значением базового типа P. Тип результата является базовым типом P. Результат представляет собой значение с номером позиции на единицу большим номера позиции для значения X. Если X равен P'BASE'LAST, то возбуждается исключение CONSTRAINT_ERROR (см. разд. 3.5.5).

P'TERMINATED (P'ЗАВЕРШЕНА). Префикс P соответствует задачному типу или задаче. Вырабатывает значение TRUE, если задача P завершена, иначе вырабатывает значение FALSE. Значение этого атрибута имеет предопределенный тип BOOLEAN (см. разд. 9.9).

P'VAL (P'ЗНАЧ). Префикс P обозначает дискретный тип или подтип. Этот атрибут является специальной функцией с одним параметром X, который может быть любого целого типа. Тип результата является базовым типом P. Результат представляет собой значение, чьим номером позиции является значение типа *универсальный_целый*, и это значение соответствует X. Если соответствующее X – *универсальное_целое* значение – не принадлежит диапазону P'POS (P'BASE'FIRST) . . P'POS (P'BASE'LAST), то возбуждается исключение CONSTRAINT_ERROR (см. разд. 3.5.5).

P'VALUE (P'ЗНАЧЕНИЕ). Префикс P обозначает дискретный тип или подтип. Этот атрибут является функцией с одним параметром. Фактический параметр X должен быть значением предопределенного типа STRING. Тип результата – базовый тип P. Любые предшествующие и последующие пробелы последовательности символов, соответствующей X, игнорируются.

Если для перечислимого типа последовательность символов имеет синтаксис литерала перечисления и если этот литерал существует для базового типа P, то результатом является соответствующее значение перечислимого типа. Если для целого типа последовательность символов имеет синтаксис целого литерала, возможно с дополнительным символом плюс или минус, и если соответствующее значение принадлежит базовому типу P, то результатом является именно это значение. Во всех остальных случаях возбуждается исключение CONSTRAINT_ERROR (см. разд. 3.5.5).

P'WIDTH (P'ШИРИНА). Префикс P обозначает дискретный подтип. Вырабатывает максимальную длину образа по всем значениям подтипа P. (*Образ* – это последовательность символов, вырабатываемых атрибутом IMAGE.) Значение этого атрибута имеет *универсальный_целый* тип (см. разд. 3.5.5).

ПРИЛОЖЕНИЕ 2
Обязательное

ПРАГМЫ, ПРЕДОПРЕДЕЛЕННЫЕ В ЯЗЫКЕ

Ниже определяются прагмы LIST, PAGE и OPTIMIZE и остальные предопределенные прагмы, которые рассматривались в описании языка. (В скобках указан пояснительный перевод.)

CONTROLLED (УПРАВЛЯЕМЫЙ). Единственным аргументом является простое имя ссылочного типа. Эта прагма допустима только непосредственно в разделе описаний или в спецификации пакета, содержащих описание этого ссылочного типа, причем описание должно помещаться перед прагмой. Не допускается использование этой прагмы для производного типа. Прагма указывает, что для объектов, указанных значениями соответствующего ссылочного типа, не должно выполняться автоматическое освобождение памяти, исключая выход из самого вложенного оператора блока, тела подпрограммы или тела задачи, охватывающих описание ссылочного типа, либо выход из главной программы (см. разд. 4.8).

ELABORATE (ПРЕДВЫПОЛНИТЬ). В качестве аргументов используются одно или несколько простых имен, обозначающих библиотечные модули. Эта прагма допустима только непосредственно после спецификатора контекста компилируемого модуля (перед библиотечным или вторичным модулем). Каждый аргумент прагмы должен быть простым именем библиотечного модуля, упомянутого в спецификаторе контекста. Эта прагма указывает, что тело соответствующего библиотечного модуля должно быть предвыполнено до предвыполнения данного компилируемого модуля. Если данный компилируемый модуль является субмодулем, то тело библиотечного модуля должно быть предвыполнено до предвыполнения тела родительского (по отношению к этому субмодулю) библиотечного модуля (см. разд. 10.5).

INLINE (ПОДСТАВИТЬ). В качестве аргументов используется одно или несколько имен; каждое имя является либо именем подпрограммы, либо именем настраиваемой подпрограммы. Эта прагма допустима только на месте элемента описания в разделе описаний или в спецификации пакета, либо же после библиотечного модуля в компиляции, но до любого следующего компилируемого модуля. Прагма указывает на возможную подстановку тела подпрограммы вместо каждого его вызова; в случае настраиваемой подпрограммы, эта прагма относится к вызовам ее конкретизаций (см. разд. 6.3.2).

INTERFACE (СВЯЗАТЬ). В качестве аргументов используются имя языка и имя подпрограммы. Эта прагма допустима на месте элемента описания и должна применяться по отношению к подпрограмме, описанной ранее в виде элемента описания того же самого раздела описаний или спецификации пакета. Эта прагма допустима также для библиотечного модуля; в этом случае прагма должна помещаться после описания подпрограммы и перед любым следующим компилируемым модулем. Эта прагма указывает другой язык (и тем самым соглашения по вызову) и информирует компилятор о том, что для соответствующей подпрограммы будет представлен объектный модуль (см. разд. 13.9).

LIST (ЛИСТИНГ). В качестве единственного аргумента используется один из идентификаторов ON (ВКЛ) или OFF (ВЫК). Эта прагма допустима в любом месте, где допустимы прагмы. Прагма указывает, что необходимо продолжить или прекратить вывод листинга компиляции до тех пор, пока в той же компиляции не встретится прагма LIST с другим аргументом. Текст самой прагмы печатается всегда, если компилятор выводит листинг.

MEMORY_SIZE (РАЗМЕР_ПАМЯТИ). В качестве единственного аргумента используется числовой литерал. Эта прагма допустима только в начале компиляции, до ее первого компилируемого модуля (если он есть). Применение этой прагмы приводит к использованию указанного числового литерала для определения именованного числа MEMORY_SIZE (см. разд. 13.7).

OPTIMIZE (ОПТИМИЗИРОВАТЬ). В качестве единственного аргумента используется один из идентификаторов TIME (ВРЕМЯ) или SPACE (ПАМЯТЬ). Эта прагма допустима только внутри раздела описаний и относится к блоку или телу, охватывающему этот раздел описаний. Она указывает главный критерий оптимизации – время выполнения или занимаемую память.

PACK (УПАКОВЫВАТЬ). В качестве единственного аргумента используется простое имя именуемого или индексируемого типа. Допустимое положение этой прагмы в программе и ограничения, относящиеся к именуемому типу, определяются теми же правилами, что и для спецификатора представления. Прагма указывает, что при выборе представления значений данного типа главным критерием оптимизации обязан быть минимум занимаемой памяти (см. разд. 13.1).

PAGE (СТРАНИЦА). Эта прагма не имеет аргументов и допустима везде, где допустимы прагмы. Она указывает, что текст программы, следующий за прагмой, должен начинаться с новой страницы (если компилятор параллельно выводит листинг).

PRIORITY (ПРИОРИТЕТ). Единственным аргументом этой прагмы является статическое выражение предопределенного целого подтипа PRIORITY. Эта прагма

допустима только в спецификации задачи или непосредственно внутри самого внешнего раздела описаний главной программы. Она указывает приоритет этой задачи (или задач этого задачного типа) или приоритет главной программы (см. разд. 9.8).

SHARED (РАЗДЕЛЕННАЯ). В качестве единственного аргумента этой прагмы используется простое имя переменной. Эта прагма допустима только для переменной, описанной посредством описания объекта скалярного или ссылочного типа. И описание переменной и прагма должны находиться (в указанном порядке) в одном и том же разделе описаний или в спецификации пакета. Эта прагма указывает, что каждое чтение или изменение значения этой переменной является ее точкой синхронизации. Реализация должна ограничивать круг объектов, для которых допустимо использование этой прагмы теми, прямое чтение или прямое изменение значения которых реализуются как неделимые операции (см. разд. 9.11).

STORAGE_UNIT (КВАНТ_ПАМЯТИ). В качестве единственного аргумента используется числовой литерал. Эта прагма допустима только в начале компиляции, до первого ее компилируемого модуля (если он есть). Применение этой прагмы приводит к использованию для определения именованного числа STORAGE_UNIT значения заданного числового литерала (см. разд. 13.7).

SUPPRESS (ПОДАВИТЬ). В качестве аргументов используются идентификатор проверки, а также имя объекта, типа или подтипа, подпрограммы, задачного модуля или настраиваемого модуля. Эта прагма допустима только непосредственно либо в разделе описаний, либо в спецификации пакета. В последнем случае единственным допустимым вариантом является использование имени, обозначающего понятие (или несколько совмещенных подпрограмм), описанное непосредственно внутри этой спецификации. Разрешение подавить указанную проверку действует от прагмы до конца области действия описаний, соответствующей самому вложенному оператору блока или программному модулю. Для прагмы, используемой в спецификации пакета, это разрешение действительно до конца области действия описания заданного своим именем понятия. Если в качестве аргумента прагмы используется имя, то разрешение на подавление указанной проверки ограничивается следующими правилами: это разрешение распространяется только на операции над указанными объектами базового типа указанного типа или подтипа, на вызовы указанной подпрограммы, на активизацию задач указанного задачного типа, на конкретизацию указанного настраиваемого модуля (см. разд. 11.7).

SYSTEM_NAME (ИМЯ_СИСТЕМЫ). Единственным аргументом прагмы является литерал перечисления. Эта прагма допустима только в начале компиляции, до первого ее компилируемого модуля (если он есть). Применение прагмы приводит к использованию литерала перечисления, заданного идентификатором, для определения константы SYSTEM_NAME. Прагма допустима только в том случае, если указанный в ней идентификатор соответствует одному из литералов перечисления типа NAME, описанного в пакете SYSTEM (см. разд. 13.7).

ПРИЛОЖЕНИЕ 3
Обязательное

ПРЕДОПРЕДЕЛЕННОЕ ОКРУЖЕНИЕ ЯЗЫКА

Ниже дана характеристика спецификации пакета STANDARD, содержащая все предопределенные в языке идентификаторы. Соответствующее тело пакета определяется реализацией и не показано.

Предопределенные для описанных в пакете STANDARD типов операции даны в виде комментариев, так как они описаны неявно. Для псевдоимен анонимных типов (*универсальный_вещественный*, например), а также для неопределенной информации (например, *определен_реализацией* и *любой_фиксированный_тип*) используется курсив.

```
package STANDARD is
    type BOOLEAN is (FALSE, TRUE);
    - - Предопределенными операциями отношения для этого типа являются
    - - следующие;
    - - function "=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
    - - function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
    - - function "<" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
    - - function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
    - - function ">" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
    - - function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
    - - Предопределенными логическими операциями для этого типа являются
    - - следующие:
    - - function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
    - - function "or" (LEFT, RIGHT : BOOLEAN) retutn BOOLEAN;
    - - function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
    - - function "not" (RIGHT : BOOLEAN) return BOOLEAN;
    - - Предопределен универсальный тип универсальный_целый
type INTEGER is определен_реализацией;
    - - Для этого типа предопределены следующие операции:
    - - function "=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
    - - function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
    - - function "<" (LEFT, RIGHT : INTEGER) return BOOLEAN;
    - - function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
    - - function ">" (LEFT, RIGHT : INTEGER) return BOOLEAN;
    - - function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
    - - function "+" (RIGHT : INTEGER) return INTEGER;
    - - function "–" (RIGHT : INTEGER) return INTEGER;
    - - function "abs" (RIGHT : INTEGER) return INTEGER;
    - - function "+" (LEFT, RIGHT : INTEGER) return INTEGER;
    - - function "–" (LEFT, RIGHT : INTEGER) return INTEGER;
    - - function "*" (LEFT, RIGHT : INTEGER) return INTEGER;
    - - function "/" (LEFT, RIGHT : INTEGER) return INTEGER;
    - - function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
    - - function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;
    - - function "**" (LEFT : INTEGER; RIGHT : INTEGER) return INTEGER;
    - - Реализация может обеспечить другие предопределенные целые типы. Рекомен-
    - - дуется, чтобы имена этих дополнительных типов оканчивались на INTEGER,
    - - например, SHORT_INTEGER или LONG_INTEGER. Спецификация каждой
    - - операции для типа универсальный_целый или для любого дополнительного
    - - предопределенного целого типа может быть получена заменой идентификатора
    - - INTEGER на имя этого типа в спецификации соответствующей операции для
    - - типа INTEGER, исключая правый операнд операции возведения в степень.
    - - Универсальный тип универсальный_вещественный предопределен
    type FLOAT is определен_реализацией;
    - - Для этого типа предопределены следующие операции:
    - - function "=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
    - - function "/=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
    - - function "<" (LEFT, RIGHT : FLOAT) return BOOLEAN;
    - - function "<=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
    - - function ">" (LEFT, RIGHT : FLOAT) return BOOLEAN;
    - - function ">=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
    - - function "+" (RIGHT : FLOAT) return FLOAT;
    - - function "–" (RIGHT : FLOAT) return FLOAT;
    - - function "abs" (RIGHT : FLOAT) return FLOAT;
    - - function "+" (LEFT, RIGHT : FLOAT) return FLOAT;
```

```
-- function "–" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "**" (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;
-- Реализация может обеспечить дополнительные предопределенные плавающие
-- типы. Рекомендуется, чтобы имена таких дополнительных типов оканчива-
-- лись на FLOAT, например, SHORT_FLOAT или LONG_FLOAT. Спецификация каждой операции для типа универсальный_вещественный или для любого дополнительного предопределенного плавающего типа может быть получена заменой идентификатора FLOAT на имя этого типа в спецификации соответствующей операции для типа FLOAT.
-- Кроме того, для универсальных типов предопределены следующие операции:
-- function "*" (LEFT : универсальный_целый; RIGHT : универсальный_вещественный)
--                 return универсальный_вещественный;
-- function "*" (LEFT : универсальный_вещественный; RIGHT : универсальный_целый)
--                 return универсальный_вещественный;
-- function "/" (LEFT : универсальный_вещественный; RIGHT : универсальный_целый)
--                 return универсальный_вещественный;
-- Тип универсальный_фиксированный предопределен. Единственными операциями, описанными для этого типа, являются:
-- function "*" (LEFT : любой_фиксированный_тип; RIGHT : любой_фиксированный_тип)
--               return универсальный_фиксированный;
-- function "/" (LEFT : любой_фиксированный_тип; RIGHT : любой_фиксированный_тип)
--               return универсальный_фиксированный;
-- Следующие символы образуют стандартный набор символов ASCII и ГОСТ
-- 27465–87. Символьные литералы, соответствующие управляющим символам,
-- идентификаторами не являются.
-- В этом определении они выделены курсивом. Символьные литералы латинского алфавита также выделены курсивом, чтобы отличить их от символьных литералов русского алфавита.
type CHARACTER is
   (nul,  soh,  stx,  etx,   eot,  enq,  ack,  bel,
    bs,   ht,   lf,   gvt,   ff,   cr,   sv,   sl,
    dle,  dcl,  dc2,  dc3,   dc4,  nak,  syn,  etb,
    can,  em,   sub,  esc,   fs,   gs,   rs,   us,
    ' ',  '!',  '"',  '#',   '$',  '%',  '&',  ''',
    '(',  ')',  '*',  '+',   ',',  '-',  '.',  '/',
    '0',  '1',  '2',  '3',   '4',  '5',  '6',  '7',
    '8',  '9',  ':',  ';',   '<',  '=',  '>',  '?',
    '@',  'A',  'B',  'C',   'D',  'E',  'F',  'G',
    'H',  'I',  'J',  'K',   'L',  'M',  'N',  'O',
    'P',  'Q',  'R',  'S',   'T',  'U',  'V',  'W',
    'X',  'Y',  'Z',  '[',   '\',  ']',  '^',  '_',
    '`',  'a',  'b',  'c',   'd',  'e',  'f',  'g',
    'h',  'i',  'j',  'k',   'l',  'm',  'n',  'o',
    'p',  'q',  'r',  's',   't',  'u',  'v',  'w',
    'x',  'y',  'z',  '{',   '|',  '}',  '~',  del);
```

```
    'А', 'Б', 'В', 'Г', 'Д', 'Е', 'Ё', 'Ж',
    'З', 'И', 'Й', 'К', 'Л', 'М', 'Н', 'О',
    'П', 'Р', 'С', 'Т', 'У', 'Ф', 'Х', 'Ц',
    'Ч', 'Ш', 'Щ', 'Ъ', 'Ы', 'Ь', 'Э', 'Ю',
    'Я', 'а', 'б', 'в', 'г', 'д', 'е', 'ё',
    'ж', 'з', 'и', 'й', 'к', 'л', 'м', 'н',
    'о', 'п', 'р', 'с', 'т', 'у', 'ф', 'х',
    'ц', 'ч', 'ш', 'щ', 'ъ', 'ы', 'ь', 'э',
    'ю', 'я', 'зб');
for CHARACTER use - - 256 символов набора
(0, 1, 2, 3, 4, 5, . . . , 193, 194, 255);
- - Предопределенные операции для типа CHARACTER те же самые, что и для
- - любого перечислимого типа.
package ASCII is
- - Управляющие символы
   NUL : constant CHARACTER : = nul; - - пусто
   SOH : constant CHARACTER : = soh; - - начало заголовка
   STX : constant CHARACTER : = stx; - - начало текста
   ETX : constant CHARACTER : = etx; - - конец текста
   EOT : constant CHARACTER : = eot; - - конец передачи
   ENQ : constant CHARACTER : = enq; - - кто там?
   ACK : constant CHARACTER : = ack; - - подтверждение
   BEL : constant CHARACTER : = bel; - - звонок
   BS : constant CHARACTER : = bs; - - возврат на шаг
   HT : constant CHARACTER : = ht; - - горизонтальная табуляция
   LF : constant CHARACTER : = lf; - - перевод строки
   VT : constant CHARACTER : = vt; - - вертикальная табуляция
   FF : constant CHARACTER : = ff; - - перевод формата
   CR : constant CHARACTER : = cr; - - возврат каретки
   SO : constant CHARACTER : = so; - - выход
   SI : constant CHARACTER : = si; - - вход
   DLE : constant CHARACTER : = dle; - - авторегистр 1
   DC1 : constant CHARACTER : = dc1; - - символы устройства
   DC2 : constant CHARACTER : = dc2;
   DC3 : constant CHARACTER : = dc3;
   DC4 : constant CHARACTER : = dc4, - - стоп
   NAK : constant CHARACTER : = nak; - - отрицание
   SYN : constant CHARACTER : = syn; - - синхронизация
   ETB : constant CHARACTER : = etb; - - конец блока
   CAN : constant CHARACTER : = can; - - аннулирование
   EM : constant CHARACTER : = em; - - конец носителя
   SUB : constant CHARACTER : = sub; - - замена
   ESC : constant CHARACTER : = esc; - - авторегистр 2
   FS : constant CHARACTER : = fs; - - разделитель файлов
   GS : constant CHARACTER : = gs; - - разделитель групп
   RS : constant CHARACTER : = rs; - - разделитель записей
   US : constant CHARACTER : = us; - - разделитель элементов
   DEL : constant CHARACTER : = del; - - забой
   - - Другие символы:
   EXCLAM : constant CHARACTER : = '!'; - - восклицательный знак
   QUOTATION : constant CHARACTER: = '"'; - - кавычки
   SHARP : constant CHARACTER : = '#'; - - номер
   DOLLAR : constant CHARACTER : = '¤'; - - знак денежной единицы
```

```
  PERCENT : constant CHARACTER : = '%'; - - процент
  AMPERSAND : constant CHARACTER : = '&'; - - коммерческое И
  COLON : constant CHARACTER : = ':'; - - двоеточие
  SEMICOLON : constant CHARACTER : = ';'; - - точка с запятой
  QUERY : constant CHARACTER : = '?'; - - вопросительный знак
  AT_SIGN : constant CHARACTER : = @; - - коммерческое ЭТ
  L_BRACKET : constant CHARACTER : = '['; - - левая квадратная скобка
  BACK_SLASH : constant CHARACTER : = '\'; - - обратная дробная черта
  BRACKET : constant     CHARACTER : = ']'; - - правая квадратная скобка
  CIRCUMFLEX : constant CHARACTER : = '^ '; - - стрелка вверх
  UNDERLINE : constant CHARACTER : = '_'; - - подчеркивание
  GRAVE : constant CHARACTER : = '`'; - - слабое ударение
  L_BRACE : constant CHARACTER : = '{'; - - левая фигурная скобка
  BAR : constant CHARACTER : = '|'; - - вертикальная черта
  R_BRACE : constant CHARACTER : = '}'; - - правая фигурная скобка
  TILDE : constant CHARACTER : = '~'; - - черта сверху
  - - Строчные буквы:
  LC_A : constant CHARACTER : = 'a';
  . . .
  LC_Z : constant CHARACTER : = 'z';
end ASCII;
package ГОСТ is
  - - управляющие символы
  ПУС : constant CHARACTER : = nul; - - пусто
  НЗ : constant CHARACTER : = soh; - - начало заголовка
  НТ : constant CHARACTER : = stx; - - начало текста
  КТ : constant CHARACTER : = etx; - - конец текста
  КП : constant CHARACTER : = eot; - - конец передачи
  КТМ : constant CHARACTER : = enq; - - кто там?
  ДА : constant CHARACTER : = ask; - - подтверждение
  ЗВ : constant CHARACTER : = bel; - - звонок
  ВШ : constant CHARACTER : = bs; - - возврат на шаг
  ГТ : constant CHARACTER : = ht; - - горизонтальная табуляция
  ПС : constant CHARACTER : = lf; - - перевод строчки
  ВТ : constant CHARACTER : = vt; - - вертикальная табуляция
  ПФ : constant CHARACTER : = ff; - - перевод формата
  ВК : constant CHARACTER : = cr; - - возврат каретки
  ВЫХ : constant CHARACTER : = so; - - выход
  ВХ : constant CHARACTER : = si; - - вход
  АР1 : constant CHARACTER : = dle; - - авторегистр 1
  СУ1 : constant CHARACTER : = dc1; - - символы устройства
  СУ2 : constant CHARACTER : = dc2;
  СУ3 : constant CHARACTER : = dc3;
  СУ4 : constant CHARACTER : = dc4; - - стоп
  НЕТ : constant CHARACTER : = nak; - - отрицание
  СИН : constant CHARACTER : = syn; - - синхронизация
  КБ : constant CHARACTER : = etb; - - конец блока
  АН : constant CHARACTER : = can; - - аннулирование
  КН : constant CHARACTER : = em; - - конец носителя
  ЗМ : constant CHARACTER : = sub; - - замена
  АР2 : constant CHARACTER : = esc; - - авторегистр 2
  РФ : constant CHARACTER : = fs; - - разделитель файлов
  РГ : constant CHARACTER : = gs; - - разделитель групп
  РЗ : constant CHARACTER : = rs; - - разделитель записей
  РЭ : constant CHARACTER : = us; - - разделитель элементов
```

```
   ЗБ : constant CHARACTER : = зб; - - забой
   - - другие символы:
   ВОСКЛ_ЗНАК : constant CHARACTER : = '!';   - - восклицательный знак
   КАВЫЧКИ : constant CHARACTER : = '"'; - - кавычки
   НОМЕР : constant CHARACTER : = '#'; - - номер
   ДЕНЕЖНЫЙ_ЗНАК : constant CHARACTER : = '¤'; - - знак денежной единицы
   ПРОЦЕНТЫ : constant CHARACTER : = '%'; - - проценты
   АМПЕРСАНД : constant CHARACTER : = '&'; - - коммерческое И
   ДВОЕТОЧИЕ : constant CHARACTER : = ':'; - - двоеточие
   ТОЧКА_С_ЗАПЯТОЙ : constant CHARACTER : = ';'; - - точка с запятой
   ВОПРОС_ЗНАК : constant CHARACTER : = '?'; - - вопросительный знак
   КОММЕРЧ_ЭТ : constant CHARACTER : = '@'; - - коммерческое ЭТ
   Л_СКОБКА : constant CHARACTER : = '['; - - левая квадратная скобка
   ОБР_ЧЕРТА : constant CHARACTER : = '\'; - - обратная дробная черта
   П_СКОБКА : constant CHARACTER : = ']'; - - правая квадратная скобка
   СТРЕЛКА_ВВЕРХ : constant CHARACTER : = '^'; - - стрелка вверх
   ПОДЧЕРК : constant CHARACTER : = '_'; - - подчеркивание
   СЛАБОЕ_УДАРЕНИЕ : constant CHARACTER : = '`'; - - слабое ударение
   ЛФ_СКОБКА : constant CHARACTER : = '{'; - - левая фигурная скобка
   ВЕРТИК_ЧЕРТА : constant CHARACTER : = '|'; - - вертикальная черта
   ПФ_СКОБКА : constant CHARACTER : = '}'; - - правая фигурная скобка
   НАДЧЕРК : constant CHARACTER : = '~'; - - черта сверху (надчеркивание)
   - - Строчные русские буквы:
   LR_A : constant CHARACTER : = 'а';
   . . .
   LR_Я : constant CHARACTER : = 'я';
end ГОСТ;
- - Предопределенные подтипы:
subtype NATURAL is INTEGER range 0 . . INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 . . INTEGER'LAST;
- - Предопределенный строковый тип:
type STRING is array (POSITIVE range < > ) of CHARACTER;
pragma PACK (STRING);
- - Следующие операции для этого типа предопределены:
- - function "=" (LEFT, RIGHT : STRING) return BOOLEAN;
- - function "/=" (LEFT, RIGHT : STRING) return BOOLEAN;
- - function "<" (LEFT, RIGHT : STRING) return BOOLEAN;
- - function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
- - function ">" (LEFT, RIGHT : STRING) return BOOLEAN;
- - function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;
- - function "&" (LEFT : STRING; RIGHT : STRING) return STRING;
- - function "&" (LEFT : CHARACTER; RIGHT : STRING) return STRING;
- - function "&" (LEFT : STRING; RIGHT : CHARACTER) reuturn STRING;
- - function "&" (LEFT : CHARACTER; RIGHT : CHARACTER) return STRING;
- - type DURATION is delta определяется_реализацией range
- - определяется_реализацией;
- - Для типа DURATION предопределены все те операции, что и для любого фик-
- - сированного типа.
- - Предопределены следующие исключения:
CONSTRAINT_ERROR : exception;
NUMERIC_ERROR : exception;
PROGRAM_ERROR : exception;
STORAGE_ERROR : exception;
TASKING_ERROR : exception;
end STANDARD;
```

Некоторые аспекты предопределенных понятий не могут быть выражены в терминах самого языка. Хотя, например, перечислимый тип BOOLEAN может быть записан посредством двух литералов перечисления FALSE и TRUE, формы управления промежуточной проверкой в самом языке выражены быть не могут.

Примечание. Определением языка предопределены следующие библиотечные модули:

– пакет CALENDAR (см. разд. 9.6);
– пакет SYSTEM (см. разд. 13.7);
– пакет MACHINE_CODE (см. разд. 13.8) (если он предусмотрен);
– настраиваемая процедура UNCHECKED_DEALLOCATION (см. разд. 13.10.1);
– настраиваемая функция UNCHECKED_CONVERSION (см. разд. 13.10.2);
– настраиваемый пакет SEQUENTIAL_IO (см. разд. 14.2.3);
– настраиваемый пакет DIRECT_IO (см. разд. 14.2.5);
– пакет TEXT_IO (см. разд. 14.3.10);
– пакет IO_EXCEPTIONS (см. разд. 14.5);
– пакет LOW_LEVEL_IO (см. разд. 14.6).

ПРИЛОЖЕНИЕ 4
Обязательное

ХАРАКТЕРИСТИКИ, ЗАВИСЯЩИЕ ОТ РЕАЛИЗАЦИИ

Определение языка Ада допускает в контролируемых пределах машинную зависимость. Не допускаются машинно-зависимые расширения или ограничения синтаксиса или семантики. Машинная зависимость допускается только в определенных реализацией прагмах и атрибутах, в машинно-зависимых соглашениях, перечисленных в гл. 13, а также в ограничениях на использование спецификаторов представления.

Руководство по каждой реализации языка программирования Ада должно включать в себя такое обязательное приложение, которое описывает все характеристики, зависящие от реализации. В таком приложении для данной реализации должны быть перечислены;

1. Форма, допустимые места расположения и результат каждой зависящей от реализации прагмы.
2. Имя и тип каждого атрибута, зависящего от реализации.
3. Спецификация пакета SYSTEM (см. разд. 13.7).
4. Список всех ограничений на спецификаторы представления (см. разд. 13.1).
5. Соглашения об использовании генерируемых реализацией имен, обозначающих компоненты, зависящие от реализации (см. разд. 13.4).
6. Интерпретация выражений, появляющихся в спецификаторах адреса, включая связанные с прерыванием (см. разд. 13.5).
7. Любое ограничение на неконтролируемые преобразования (см. разд. 13.10.2):
8. Любые зависящие от реализации особенности для пакетов ввода-вывода (см. гл. 14).

ПРИЛОЖЕНИЕ 5
Справочное

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Это приложение носит информационный характер и не является частью определения стандарта языка. Термины, выделенные в тексте курсивом, либо имеют собственную статью, либо описываются в статьях, относящихся к логически связанным с ними понятиям. После термина в скобках приведен соответствующий английский термин.

Агрегат (aggregate) – базовая операция над типом, которая объединяет значения *компонентов* в составное значение *индексируемого* типа (агрегат массива) или *именуемого типа* (агрегат записи), являющихся разновидностями составного типа. Компоненты агрегата могут быть *позиционными* (координатными) и/или *именованными* (ключевыми).

Атрибут (attribute) – базовая операция над типом, которая вырабатывает предопределенную характеристику поименованного понятия, указанного *префиксом;* некоторые атрибуты являются *функциями, типом* или *диапазоном.*

Вещественный тип (real type) – набор приближенных значений вещественных чисел, заданных с относительной погрешностью (*плавающий тип*) или абсолютной погрешностью (*фиксированный тип*). Вещественный тип реализуется *хранимыми* в памяти вычислительной машины числами и определяется точностью приближения и *диапазоном.*

Вид (mode) – см. *параметр.*

Видимость (visibility) – доступность характеристик описанного понятия при его применении в данной точке текста программы. *Описание* понятия, вводящее идентификатор, называется *видимым* в данной точке текста программы, если это понятие осмысленно при появлении его идентификатора в этой точке программы. Прямая видимость подразумевает одно описание понятия с характеристиками, уникальными в *области действия*, и некоторое число использований этого понятия. При *именовании* описание *видимо* на месте *постфикса* в именуемом компоненте или на месте имени в *именованном сопоставлении.* Иначе, описание *видимо непосредственно* всякий раз, когда единственный идентификатор имеет смысл, определяемый этим описанием.

Видимый раздел (visible part) – см. *пакет.*

Возбуждение исключения (raising an exception) – см. *исключение.*

Вход (entry) – именование точки синхронизации *задач* и передачи данных между ними. Вход используется для осуществления взаимодействия задач. Синтаксически вызов входа походит на вызов *подпрограммы*; его внутренний механизм определяется одним или несколькими *операторами принятия*, специфицирующими действия, выполняемые при вызове входа.

Вычисление (evaluation) – процесс получения значения (например, *выражения*). Этот процесс происходит во время выполнения программы.

Выражение (expression) – формула, определяющая процесс получения значения.

Генератор (allocator) – базовая операция над типом, создающая *объект* и вырабатывающая *ссылочное значение,* которое указывает на этот объект. Созданный объект должен сохраняться до тех пор, пока существует обозначение объекта некоторым именем.

Диапазон (range) – упорядоченный набор последовательных значений *скалярного типа.* Диапазон задается нижней и верхней границами этого набора значений. Значение из диапазона называется *принадлежащим* этому диапазону.

Дискретный тип (discret type) – упорядоченный набор различных значений. Дискретными типами являются *перечислимый* и *целый типы.* Дискретные типы используются для индексирования и управления повторением в операторах цикла, а также в выборах операторов выбора и вариантах записи.

Дискриминант (discriminant) – специальный *компонент объекта* или значение

именуемого типа. Подтипы других компонентов и даже их присутствие или отсутствие могут зависеть от значения дискриминанта.

Задача (task) – *программный модуль*, функционирующий параллельно с другими частями программы. Задача задается *спецификацией задачи*, которая задает *имя* задачи, а также имена и *формальные параметры* ее *входов*, и *телом* задачи, которое определяет ее выполнение. *Задачный модуль (модуль-задача)* является одним из видов программных модулей. *Задачный тип* – это *тип*, который предоставляет возможность последующего описания любого количества однотипных задач. Говорят, что значение задачного типа *указывает* задачу.

Именованное сопоставление (named association) – способ задания связи элемента составного значения с одной или несколькими позициями с помощью их именования.

Именуемый компонент (selected component) – *имя*, состоящее из *префикса* и идентификатора, называемого *постфиксом*. Именуемые компоненты используются для обозначения компонентов записей, *входов* и *объектов*, указанных ссылочными значениями; они также используются как *расширенные имена*.

Именуемый тип (record type) – составной тип из именованных различными идентификаторами *компонентов*, которые обычно бывают различных *типов* или *подтипов*. Для каждого компонента значения записи или *объекта*-записи в определении именуемого типа задается идентификатор, который однозначно определяет компонент записи.

Имя (name) – средство представления понятия. Говорят, что имя *обозначает* понятие, и что понятие является смыслом имени. См. также *описание, префикс*.

Индекс (index) – см. *индексируемый тип*.

Индексируемый компонент (indexed component) – форма *имени*, содержащая *выражения*, которые задают значения *индексов компонента массива*. Индексируемый компонент *обозначает* компонент массива. Индексируемый компонент может также обозначать *вход* в семействе входов.

Индексируемый тип (array type) – составной тип из *компонентов* одного и того же *подтипа* (и, следовательно, одного и того же типа). Каждый компонент однозначно идентифицируется *индексом* (для одномерного массива) или последовательностью индексов (для многомерного массива). Каждый индекс должен быть значением *дискретного типа* и должен принадлежать требуемому диапазону индексов.

Исключение (exception) – обозначение ошибочной ситуации, которая может произойти при выполнении программы и при этом будет зарегистрирована и обработана. *Возбуждение* исключения состоит в прекращении нормального выполнения программы, сигнализирующем о наличии ошибки. *Обработчик исключения* – это резервная часть программного текста, задающая реакцию на исключение. Выполнение этого программного текста называется *обработкой исключения*.

Квалифицированное выражение (qualified expresson) – *выражение*, перед которым указан его *тип* или *подтип*. Используется для разрешения неоднозначности выражения (например, из-за *совмещения*).

Компилируемый модуль (compilation unit) – *описание* или *тело программного модуля*, предназначенные для компиляции в качестве самостоятельного текста. Перед ним может быть задан *спецификатор контекста*, включающего другие компилируемые модули, от которых зависит данный, и имена которых указаны в *спецификаторах совместности*.

Компонент (component) – значение, которое является частью более сложного значения, или *объект*, который является частью более сложного объекта.

Константа (constant) – см. *объект*.

Лексема (lexical element) – лексический элемент, который может быть идентификатором, *литералом*, ограничителем или комментарием.

Лимитируемый тип (limited type) – *тип*, для которого не определены неявно описанные операции присваивания и предопределенного сравнения на равенство. Все *задачные типы* – лимитируемые. *Личный тип* может быть определен как лимитируемый. Для лимитируемого типа может быть явно описана операция сравнения на равенство.

Литерал (literal) – значение, явно выраженное буквами, цифрами или другими символами. Литерал – это одно из четырех: числовой литерал, литерал перечисления, символьный литерал или строковый литерал.

Личный раздел (private part) – см. *пакет*.

Личный тип (private type) – *тип*, структура и набор значений которого явно определены, но непосредственно недоступны для пользователя. О личном типе известны только его *дискриминанты* (если они есть) и набор *операций*, определенных над его значениями. Личный тип и соответствующие операции определяются в *видимом разделе пакета* или в *разделе формальных параметров настройки*. Для личных типов, не являющихся *лимитируемыми*, определены также операции присваивания, сравнения на равенство и неравенство.

Модельное число (model number) – точно представляемое значение *вещественного типа*. *Операции* над вещественными типами определяются в терминах операций над модельными числами этих типов. Свойства модельных чисел и операции над ними являются минимальными свойствами, предписанными для всех реализаций вещественных чисел.

Набор (collection) – вся совокупность *объектов*, создаваемых вычислением *генераторов* для некоторого *ссылочного типа*.

Настраиваемый модуль (generic unit) – шаблон для множества *подпрограмм* или *пакетов*. Создаваемые с использованием этого шаблона подпрограмма или пакет называются *экземплярами* данного настраиваемого модуля. *Конкретизация настройки* является видом *описания*, которое создает экземпляр. Настраиваемый модуль пишется в виде подпрограммы или пакета с предшествующим спецификации *разделом формальных параметров настройки*. *Формальным параметром* настройки является либо тип, либо *подпрограмма*, либо *объект*. Настраиваемый модуль – это один из видов *программных модулей*.

Область действия (scope) – см. *описание*.

Обозначить (denote) – см. *описание*.

Обработчик (handler) – см. *исключение*.

Объект (object) – понятие, которое обладает значением некоторого *типа*. *Программа* создает объект либо при *предвыполнении описания объекта*, либо при *вычислении генератора*. Описание или генератор задают тип объекта, объект может обладать значением только этого типа.

Ограничение (constraint) – средство выделения подмножества значений *типа*. Принадлежащее этому подмножеству значение *удовлетворяет* ограничению.

Ограничение диапазона (range constraint) – способ определения *диапазона типа*, т.е. подмножества значений этого типа, принадлежащих диапазону.

Ограничение дискриминанта (discriminant constraint) – способ определения дискриминанта для *именуемого типа* или *личного типа*.

Ограничение индекса (index constraint) – определение ограничения в задании нижней и верхней границ диапазона для каждого индекса *индексируемого типа*.

Оператор (statement) – синтаксическая конструкция, определяющая одно или несколько действий, реализуемых во время выполнения *программы*.

Оператор блока (block statement) – составной оператор, который может содержать последовательность *операторов*. Он может также содержать *раздел описаний* и *обработчики исключений*, которые являются локальными в данном операторе блока.

Оператор принятия (accept statement) – см. *вход*.

Операция (operator) – ограничитель или зарезервированное слово, используемые для указания алгоритма преобразования значений одного или двух операндов в результат заданного *типа*. Унарную операцию записывают перед операндом; бинарную операцию – между двумя операндами. Операция – это специальный вид *вызова функции*. Операция может быть описана как функция. Многие операции неявно описываются *описанием типа* (например, большинство описаний типа подразумевает описание операции сравнения на равенство для значений этого типа).

Операция типа (operation) – элементарное действие, связанное с одним или нес-

колькими *типами*. Операция типа неявно описывается при описании этого типа, либо является *подпрограммой*, которая имеет *параметр* или *результат* заданного типа. Вместо термина "операция типа" в стандарте используется термин "операция", который не вызывает неоднозначности при его использовании в контексте.

Описание (declaration) – синтаксическая конструкция, которая связывает идентификатор (или другие обозначения) с понятием. Это сопоставление находится внутри области текста, называемого *областью действия* описания. Внутри области действия описания существуют мета использования идентификатора для ссылки на связанное с ним понятие. Идентификатор, употребляемый в таких местах, называется *простым именем* понятия; говорят, что *имя обозначает* связанное с ним понятие.

Описание переименования (renaming declaration) – описание другого *имени* понятия.

Пакет (package) – *программный модуль*, который определяет группу логически связанных понятий, таких как *типы*, *объекты* этих типов и *подпрограммы* с *параметрами* этих типов. Пакет состоит из *описания пакета* и *тела пакета*. Описание пакета имеет *видимый раздел*, содержащий *описания* всех понятий, которые могут быть явно использованы вне пакета. Он может содержать также *личный раздел* с деталями реализации, которые заканчивают спецификацию видимых понятий, но которые недоступны для пользователя пакета. *Тело пакета* содержит реализации *подпрограмм* (и, возможно, *задач*), которые заданы в описании пакета. Пакет – это один из видов программного модуля.

Параметр (parameter) – одно из именованных понятий, связанных с *подпрограммой*, *входом* или *настраиваемым модулем* и используемых для связи с соответствующим *телом* подпрограммы, *оператором принятия* или настраиваемым телом. *Формальный параметр* – это идентификатор, используемый для обозначения именованного понятия в теле. *Фактический параметр* – это конкретное понятие, сопоставленное соответствующему формальному параметру при *вызове подпрограммы*, *вызове входа* или *конкретизации настройки*. Вид формального параметра определяет, поставляет ли фактический параметр значение для формального или формальный параметр – для фактического, или и то и другое. Сопоставление фактических параметров формальным параметрам может быть определено с помощью *именованного сопоставления* или с помощью *позиционного сопоставления* или их комбинацией.

Переменная (variable) – см. *объект*.

Перечислимый тип (enumeration type) – *дискретный тип*, значения которого представляются *литералами* перечисления, заданными явно в *описании типа*. Эти литералы перечисления являются либо идентификаторами, либо *символьными литералами*.

Плавающий тип (floating point type) – см. *вещественный тип*.

Подкомпонент (subcomponent) – *компонент* либо компонент другого компонента или подкомпонента.

Подпрограмма (subprogram) – *программный модуль*, который может быть либо *процедурой*, либо *функцией*. Процедура определяет последовательность действий и вызывается с помощью *вызова процедуры*. Функция определяет последовательность действий, а также возвращает значение, называемое *результатом*, и, таким образом, *вызов функции* является *выражением*. Подпрограмма задается в виде *описания подпрограммы*, которое определяет ее *имя*, *формальные параметры* и (для функции) ее результат, и *тела подпрограммы*, которое определяет последовательность действий. В вызове подпрограммы задаются *фактические параметры*, которые сопоставляются с формальными параметрами. Подпрограмма – это один из видов программного модуля.

Подтип (subtype) – набор значений данного *типа*, определяемый *ограничением* типа. Каждое значение из множества значений подтипа *принадлежит* этому подтипу и *удовлетворяет* ограничению, определяющему подтип.

Позиционное сопоставление (positional association) – способ задания связи элемента с позицией, использующий позицию размещения элемента для определения этого элемента.

Постфикс (selector) – см. *именуемый компонент*.

Прагма (pragma) – языковая конструкция для передачи информации компилятору.

Предвыполнение (elaboration) – процесс, применяемый к *описанию*, в результате которого описание выполняет свое назначение (например, создается *объект*). Этот процесс происходит при выполнении *программы*.

Префикс (prefix) – начальная часть некоторых видов *имени*. Префикс – это либо *вызов функции*, либо *имя*.

Присваивание (assignement) – базовая *операция типа*, заменяющая текущее значение *переменной* новым значением. *Оператор присваивания* определяет слева переменную, а справа – *выражение*, значение которого обязано стать новым значением переменной.

Программа (program) – совокупность из нескольких *компилируемых модулей*, один из которых является подпрограммой, называемой *главной программой*. Выполнение программы состоит из выполнения главной программы, которая может вызвать подпрограммы, описанные в других компилируемых модулях программы.

Программный модуль (program unit) – либо *настраиваемый модуль*, либо *пакет*, либо *подпрограмма*, либо *задачный модуль*.

Производный тип (derived type) – тип, значения и *операции* которого есть копии значений и операций существующего типа. Существующий тип называется *родительским типом* производного типа.

Простое имя (simple name) – см. *описание, имя*

Процедура (procedure) – см. *подпрограмма*.

Прямая видимость (direct visibility) – см. *видимость*.

Раздел вариантов (variant part) – определяет *альтернативные компоненты записи* в зависимости от значения *дискриминанта* записи. Каждое значение дискриминанта устанавливает одну из альтернатив раздела вариантов.

Раздел описаний (declarative part) – последовательность *описаний*. Он может также содержать логически связанную информацию, например, *тела подпрограмм* и *спецификаторы представления*.

Рандеву (rendezvous) – взаимодействие между двумя параллельно выполняемыми *задачами*, когда одна задача вызвала *вход* другой задачи, и в вызванной задаче для этого вызова выполняется соответствующий *оператор принятия*.

Расширенное имя (expanded name) – способ *обозначения* понятия, которое *описано* непосредственно внутри некоторой конструкции. Расширенное имя имеет форму *именуемого компонента: префикс* обозначает конструкцию (*программный модуль* или *блок*, *цикл* или *оператор принятия*), *постфикс* – это *простое имя* понятия.

Родительский тип (parent type) – см. *производный тип*.

Скалярный тип (scalar type) – упорядоченный набор значений с операциями отношения. К скалярному типу относится *дискретный* или *вещественный типы*. *Объект* или значение скалярного типа не имеет *компонентов*.

Совмещение (overloading) – свойство понятия иметь несколько альтернативных назначений в данной точке программного текста. Например, совмещенным *литералом* перечисления может быть идентификатор, который появляется в определениях нескольких *перечислимых типов*. Реальный смысл совмещенного идентификатора определяется по контексту. Совмещенными могут быть также *подпрограммы*, *агрегаты*, *генераторы* и *строковые литералы*.

Составной тип (composite type) – тип, значения которого имеют *компоненты*. Существуют две разновидности составного типа: *индексируемые типы* и *именуемые типы*.

Спецификатор использования (use clause) – средство, обеспечивающее прямую видимость *описаний*, которые находятся в *видимых разделах* именованных *пакетов*.

Спецификатор контекста (context clause) – см. *компилируемый модуль*.

Спецификатор представления (representation clause) – средство указания компилятору отображения *типа*, *объекта* или *задачи* на архитектуру объектной машины, на которой выполняется *программа*. В некоторых случаях спецификаторы представления полностью определяют отображение, в других случаях они создают критерии выбора отображения.

Спецификатор совместности (with clause) – см. *компилируемый модуль*.

Ссылочный тип (access type) – набор значений (*ссылочные значения*), которые могут быть либо пустым значением, либо значением, *указывающим объект*, созданный *генератором*. Значение указанного объекта может быть прочитано или изменено через ссылочное значение. Определение ссылочного типа задает тип объектов, на которые указывают значения ссылочного типа. См. также *набор*.

Субмодуль (subunit) – см. *тело*.

Тело (body) – конструкция, определяющая процесс выполнения *подпрограммы*, *пакета* или *задачи*. *След тела* является синтаксической формой тела, которая указывает, что его выполнение определяется раздельно компилируемым *субмодулем*.

Тип (type) – набор значений и набор *операций типа*, применимых к этим значениям. *Определение типа* – это языковая конструкция, которой вводится тип. Конкретный тип – это *ссылочный тип*, *индексируемый тип*, *личный тип*, *именуемый тип*, *скалярный тип* или *задачный тип*.

Удовлетворять (satisfy) – см. *ограничение*, *подтип*.

Указывать (designate) – см. *ссылочный тип*, *задача*.

Фактический параметр (actual parameter) – см. *параметр*.

Фиксированный тип (fixed point type) – см. *вещественный тип*.

Формальный параметр (formal parameter) – см. *параметр*.

Функция (function) – см. *подпрограмма*.

Целый тип (integer type) – *дискретный тип*, значения которого представляют все целые числа в заданном *диапазоне*.

Экземпляр (instance) – см. *настраиваемый модуль*.

ПРИЛОЖЕНИЕ 6
Справочное

СВОДКА СИНТАКСИСА

2.1
графический_символ ::= основной_графический_символ
| строчная_буква | дополнительный_специальный_символ
основной_графический_символ :: = прописная_буква | цифра | специальный_символ | символ_пробела основной_символ :: = основной_графический_символ
| символ_управления_форматом
2.3
идентификатор ::= буква {[подчеркивание] буква_или_цифра}
буква_или_цифра ::= буква | цифра
буква ::= прописная_буква | строчная_буква
2.4
числовой_литерал ::= десятичный_литерал | литерал_с_основанием
2.4.1
десятичный_литерал ::= целое [.целое] [порядок]
целое ::= цифра {[подчеркивание] цифра}
порядок ::= E [+] целое | E – целое
2.4.2
литерал_с_основанием::=основание # целое_с_основанием
[. целое_с_основанием] # [порядок]
основание ::= целое
целое_с_основанием ::= расширенная_цифра
{[подчеркивание] расширенная_цифра}
расширенная_цифра ::= цифра | буква

2.5
символьный_литерал ::= ' графический_символ'
2.6
строковый_литерал ::= " {графический_символ} "
2.8
прагма ::= pragma идентификатор [(сопоставление_аргумента
{, сопоставление_аргумента})];
сопоставление_аргумента ::= [идентификатор_аргумента =>] имя
| [идентификатор_аргумента =>] выражение
3.1
основное_описание ::=
описание_объекта | описание_числа
| описание_типа | описание_подтипа
| описание_подпрограммы | описание_пакета
| описание_задачи | описание_настройки
| описание_исключения | конкретизация_настройки
| описание_переименования | описание_субконстанты
3.2
описание_объекта ::=
список_идентификаторов : [constant] указание_подтипа [:= выражение];
| список_идентификаторов :[constant]
ограниченный_индексируемый_тип [:= выражение];
описание_числа ::=
список_ идентификаторов :constant : =
*универсальное_статическое*_выражение;
список_идентификаторов::= идентификатор {, идентификатор}
3.3.1
описание_типа ::= полное_описание_типа
| неполное_описание_типа | описание_личного_типа
полное_описание_типа ::=
type идентификатор [раздел_дискриминантов]
is определение_типа;
определение_типа ::= определение_перечислимого_типа
| определение_целого_типа | определение_вещественного_типа
| определение_индексируемого_типа | определение_именуемого_типа
| определение_ссылочного_типа | определение_производного_типа
3.3.2
описание_подтипа ::= subtype идентификатор is указание_подтипа;
указание_подтипа ::= обозначение_типа [ограничение]
обозначение_типа ::= имя_*типа* | имя_ *подтипа*
ограничение ::= ограничение_диапазона
| ограничение_плавающего_типа | ограничение_фиксированного_типа
| ограничение_индекса | ограничение_дискриминанта
3.4
определение_производного_типа ::= new указание_подтипа
3.5
ограничение_диапазона ::= range диапазон
диапазон ::= атрибут_диапазона | простое_выражение . . простое_выражение
3.5.1
определение_перечислимого_типа ::= (спецификация_литерала_перечисления
{, спецификация_литерала_перечисления})
спецификация_литерала_перечисления ::= литерал_перечисления
литерал_перечисления ::= идентификатор | символьный_литерал
3.5.4
определение_целого_типа ::= ограничение_диапазона

3.5.6
определение_вещественного_типа ::=
    ограничение_плавающего_типа | ограничение_фиксированного_типа
3.5.7
ограничение_плавающего_типа ::=
    определение_точности_плавающего_типа [ограничение_диапазона]
определение_точности_плавающего_типа ::=
    digits *статическое*_простое_выражение
3.5.9
ограничение_фиксированного_типа ::=
    определение_точности_фиксированного_типа [ограничение_диапазона]
определение_точности_фиксированного_типа ::=
    delta *статическое*_простое_выражение
3.6
определение_индексируемого_типа ::=
    определение_неограниченного_индексируемого_типа
    | определение_ограниченного_индексируемого_типа
определение_неограниченного_индексируемого_типа ::=
    array (определение_подтипа_индекса
      {, определение_подтипа_индекса}) of
      указание_подтипа_*компонента*
определение_ограниченного_индексируемого_типа ::=
    array ограничение_индекса of
      указание_подтипа_*компонента*
определение_подтипа_индекса ::=
    обозначение_типа range <>
ограничение_индекса ::=
    (дискретный_диапазон {, дискретный_диапазон})
дискретный_диапазон ::=
    указание_*дискретного*_подтипа | диапазон
3.7
определение_именуемого_типа ::=
     record
       список_компонентов
     end record
список_компонентов ::=
     описание_компонента {описание_компонента}
| |{описание_компонента} раздел_вариантов | null;
описание_компонента:: =
     список_идентификаторов : определение_подтипа_компонента [:= выражение];
определение_подтипа_компонента ::= указание_подтипа
3.7.1
раздел_дискриминантов ::=
     (спецификация_дискриминанта
     {; спецификация_дискриминанта})
спецификация_дискриминанта ::=
     список_идентификаторов : обозначение_типа [:= выражение]
3.7.2
ограничение_дискриминанта ::= (сопоставление_дискриминанта
  {, сопоставление_дискриминанта })
сопоставление_дискриминанта ::= [простое_имя_*дискриминанта*
     {| простое_имя_*дискриминанта*} =>] выражение
3.7.3
раздел_вариантов ::=
     case простое_имя_*дискриминанта* is

```
        вариант
        {вариант}
      end case;
вариант ::=
      when выбор {| выбор} =>
        список_ компонентов
выбор ::= простое_выражение
      | дискретный_диапазон | others
      | простое_имя_компонента
3.8
определение_ссылочного_типа ::=
      access указание_подтипа
3.8.1
неполное_описание_типа ::=
      type идентификатор [раздел_дискриминантов] ;
3.9
раздел_описаний ::= {основной_элемент_описания}
      {дополнительный_элемент_описания}
основной_элемент_описания ::= основное_описание
      | спецификатор_представления | спецификатор_использования
дополнительный_элемент_описания ::= тело
      | описание_подпрограммы | описание_пакета
      | описание_задачи | описание_настройки
      | спецификатор_использования | конкретизация_настройки
тело ::= соответствующее_тело | след_тела
соответствующее_тело ::= тело_подпрограммы | тело_пакета | тело_задачи
4.1
имя ::= простое_имя
      | символьный_литерал | знак_операции
      | индексируемый_компонент | отрезок
      | именуемый_компонент | атрибут
простое_имя ::= идентификатор
префикс ::= имя | вызов_функции
4.1.1
индексируемый_компонент ::= префикс (выражение {, выражение})
4.1.2
отрезок ::= префикс (дискретный_диапазон)
4.1.3
именуемый_компонент ::= префикс. постфикс
постфикс ::= простое_имя | символьный_литерал | знак_операции | all
4.1.4
атрибут ::= префикс' обозначение_атрибута
обозначение_атрибута ::= простое_имя [(универсальное_статическое_выражение)]
4.3
агрегат ::= (сопоставление_компонентов {, сопоставление_компонентов})
сопоставление_компонентов ::= [выбор {| выбор} =>] выражение
4.4
выражение ::= отношение {and отношение}
      | отношение {and then отношение} | отношение {or отношение}
      | отношение {or else отношение} | отношение {xor отношение}
отношение ::=
      простое_выражение [операция_отношения простое_выражение]
      | простое_выражение [not] in диапазон
      | простое_выражение [not] in обозначение_типа
простое_выражение ::=
```

```
      [унарная_аддитивная_операция] слагаемое
      {бинарная_аддитивная_операция слагаемое}
слагаемое ::=
      множитель {мультипликативная_операция множитель}
множитель ::= первичное [** первичное] | abs первичное | not первичное
первичное ::= числовой_литерал | null | агрегат | строковый_литерал | имя | генератор
      | вызов_функции | преобразование_типа | квалифицированное_выражение |
      | (выражение)
4.5
логическая_операция ::= and | or | xor
операция_отношения ::= = | /= | < | <= | > | >=
бинарная_аддитивная_операция ::= + | - | &
унарная_аддитивная_операция :: = + | -
мультипликативная_операция ::= * | / | mod | rem
операция_высшего_приоритета :: = ** | abs | not
4.6
преобразование_типа ::= обозначение_типа (выражение)
4.7
квалифицированное_выражение ::=
      обозначение_типа' (выражение) | обозначение_типа' агрегат
4.8
генератор ::= new указатель_подтипа | new квалифицированное_выражение
5.1
последовательность_операторов ::= оператор {оператор}
оператор ::= {метка} простой_оператор | {метка} составной_оператор
простой_оператор ::= пустой_оператор
      | оператор_присваивания | оператор_вызова_процедуры
      | оператор_выхода | оператор_возврата
      | оператор_перехода | оператор_вызова_входа
      | оператор_задержки | оператор_прекращения
      | оператор_возбуждения | оператор_кода
составной_оператор ::=
      условный_оператор | оператор_выбора
      | оператор_цикла | оператор_блока
      | оператор_принятия | оператор_отбора
метка ::=<<простое_имя_метки>> :
пустой_оператор ::= null;
5.2
оператор_присваивания ::= имя_переменной := выражение;
5.3
условный_оператор ::=
      if условие then
        последовательность_операторов
      {els if условие then
        последовательность_операторов}
      [else
        последовательность_операторов]
      end if;
условие ::= логическое_выражение
5.4
оператор_выбора ::=
      case выражение is
        альтернатива_оператора_выбора
        {альтернатива_оператора_выбора}
      end case;
альтернатива_оператора_выбора :: =
      when выбор {| выбор} = >
```

```
      последовательность_операторов
5.5
оператор_цикла :: =
     [простое_имя_цикла :]
       [схема_итерации] loop
          последовательность_операторов
       end loop [простое_имя_цикла] ;
схема_итерации :: = while условие | for спецификация_параметра_цикла
спецификация_параметра_цикла : : =
     идентификатор in [reverse] дискретный_диапазон
5.6
оператор_блока :: =
     [простое_имя_блока :]
       [declare
          раздел_описаний]
        begin
          последовательность_операторов
       [exception
           обработчик_исключения
          {обработчик_исключения}]
       end [простое_имя_блока] ;
5.7
оператор_выхода :: = exit [имя_цикла] [when условие] ;
5.8
оператор_возврата :: = return [выражение] ;
5.9
оператор_перехода :: = goto имя_метки;
6.1
описание_подпрограммы ::= спецификация_подпрограммы;
спецификация_подпрограммы :: =
     procedure идентификатор [раздел_формальных_параметров]
     | function обозначение [раздел_формальных_параметров ]
          return обозначение_типа
обозначение :: = идентификатор | знак_операции
знак_операции :: = строковый_литерал
раздел_формальных_параметров :: =
      (спецификация_параметра {; спецификация_параметра})
спецификация_параметра :: = список_идентификаторов : вид обозначение_типа
[: = выражение]
вид :: = [in] | in out | out
6.3
тело_подпрограммы :: =
     спецификация_подпрограммы is
       [раздел_описаний]
     begin
       последовательность_операторов
      [exception
       обработчик_исключения
       {обработчик_исключения}]
     end [обозначение] ;
6.4
оператор_вызова_процедуры :: =
      имя_процедуры [раздел_фактических_параметров] ;
вызов_функции :: = имя_функции [раздел_фактических_параметров]
раздел_фактических_параметров :: = (сопоставление_параметров
```

{ , сопоставление_параметров})
сопоставление_параметров :: =
[формальный_параметр = >] фактический_параметр
формальный_параметр :: = простое_имя_*параметра*
фактический_параметр :: = выражение | имя_*переменной*
| обозначение_типа (имя_*переменной)*
7.1
описание_пакета :: = спецификация_пакета;
спецификация_пакета :: =
package идентификатор **is**
{основной_элемент_описания}
[**private**]
{основной_элемент_описания}]
end [простое_имя_*пакета*]
тело_пакета :: =
package body простое_имя_*пакета* **is**
[раздел_описаний]
[**begin**
последовательность_операторов
[**exception**
обработчик_ сключения
{обработчик_исключения}]]
end [простое_имя_*пакета*] ;
7.4
описание_личного_типа :: =
type идентификатор [раздел_дискриминантов] **is** [**limited**] **private**;
описание_субконстанты :: = список_идентификаторов : **constant** обозначение_типа;
8.4
спецификатор_использования :: = **use** имя_*пакета* {, имя_*пакета*};
8.5
описание_переименования :: =
идентификатор : обозначение_типа **renames** имя_*объекта;*
| идентификатор : **exception renames** имя_*исключения;*
| **package** идентификатор **renames** имя_*пакета;*
| спецификация_подпрограммы **renames**
имя_*подпрограммы_или_входа;*
9.1
описание_задачи :: = спецификация_задачи;
спецификация_задачи :: =
task [**type**] идентификатор [**is**
{описание_входа}
{спецификатор_представления}
end [простое_имя_*задачи*]] ;
тело_задачи :: =
task body простое_имя_*задачи* **is**
[раздел_описаний]
begin
последовательность_операторов
[**exception**
обработчик_исключения
{обработчик_исключения}]
end [простое_имя_*задачи*] ;
9.5
описание_входа :: =
entry идентификатор [(дискретный_диапазон)]

```
        [раздел_формальных_параметров];
оператор_вызова_входа :: = имя_входа [раздел_фактических_параметров];
оператор_принятия :: =
    accept простое_имя_входа [(индекс_входа)]
        [раздел_формальных_параметров] [do
          последовательность_операторов
    end [простое_имя_входа] ];
индекс_входа :: = выражение
```

9.6

```
оператор_задержки :: = delay простое_выражение;
```

9.7

```
оператор_отбора :: = отбор_с_ожиданием | условный_вызов_входа
    | временной_вызов_входа
```

9.7.1

```
отбор_с_ожиданием :: =
    select
      альтернатива_отбора
    {or
      альтернатива_отбора}
    [else
      последовательность_операторов]
    end select;
альтернатива_отбора :: =
    [when условие = >]
      альтернатива_отбора_с_ожиданием
альтернатива_отбора_с_ожиданием :: = альтернатива_принятия
    | альтернатива_задержки | альтернатива_завершения
альтернатива_принятия :: = оператор_принятия [последовательность_операторов]
альтернатива_задержки :: = оператор_задержки [последовательность_операторов]
альтернатива_завершения :: = terminate;
```

9.7.2

```
условный_вызов_входа :: =
    select
      оператор_вызова_входа
      [последовательность_операторов]
    else
      последовательность_операторов
    end select;
```

9.7.3

```
временной_вызов_входа :: =
    select
      оператор_вызова_входа
      [последовательность_операторов]
    or
      альтернатива_задержки
    end select;
```

9.10

```
оператор_прекращения :: = abort имя_задачи {, имя_задачи};
```

10.1

```
компиляция :: = {компилируемый_модуль}
компилируемый_модуль :: =спецификатор_контекста библиотечный_модуль
    | спецификатор_контекста вторичный_модуль
библиотечный_модуль :: = описание_подпрограммы
    | описание_пакета | описание_настройки
    | конкретизация_настройки | тело_подпрограммы
```

вторичный_модуль :: = тело_библиотечного_модуля | субмодуль
тело_библиотечного_модуля :: = тело_подпрограммы | тело_пакета
10.1.1
спецификатор_контекста :: =
{ спецификатор_совместности {спецификатор_использования}}
спецификатор_совместности :: =
with простое_имя_*модуля* {, простое_имя_*модуля*};
10.2
след_тела :: = спецификация_подпрограммы is separate;
| package body простое_имя_*пакета* is separate;
| tack body простое_имя_*задачи* is separate;
субмодуль :: = separate (имя_*родительского_модуля*) соответствующее_тело
11.1
описание_исключения :: = список_идентификаторов : exception;
11.2
обработчик_исключения :: =
when выбор_исключения {| выбор_исключения} = >
последовательность_операторов
выбор_исключения :: = имя_*исключения* | others
11.3
оператор_возбуждения :: = raise [имя_*исключения*];
12.1
описание_настройки :: = спецификация_настройки;
спецификация_настройки :: =
раздел_формальных_параметров_настройки спецификация_подпрограммы
| раздел_формальных_параметров_настройки спецификация_пакета
раздел_формальных_ параметров_настройки :: =
generic {описание_параметра_настройки}
описание_параметра_настройки :: =
список_идентификаторов:
[in [out]] обозначение_типа [: = выражение];
| type идентификатор is определение_настраиваемого_типа;
| описание_личного_типа
| with спецификация_подпрограммы [is имя];
| with спецификация_подпрограммы [is < >];
определение_настраиваемого_типа :: =
(< >) | range < > | digits < > | delta < >
| определение_индексируемого_типа | определение ссылочного_типа
12.3
конкретизация_настройки :: =
package идентификатор is new имя_*настраиваемого_пакета*
[раздел_фактических_параметров_настройки];
| procedure идентификатор is new имя_*настраиваемой_процедуры*
[раздел_фактических_параметров_настройки];
| function обозначение is new имя_*настраиваемой_функции*
[раздел_фактических_параметров_настройки];
раздел_фактических_параметров_настройки :: =
(сопоставление_параметров_настройки
{, сопоставление_параметров_настройки})
сопоставление_параметров_настройки :: =
[формальный_параметр_настройки = >] фактический_параметр_настройки
формальный_параметр_настройки :: =
простое_имя_*параметра* | знак_операции
фактический_параметр_настройки :: = выражение
| имя_*переменной* | имя_*подпрограммы*

```
    | имя_входа | обозначение_типа
13.1
спецификатор_представления :: =
    спецификатор_представления_типа | спецификатор_адреса
спецификатор_представления_типа :: = спецификатор_длины
    | спецификатор_представления_перечисления
    | спецификатор_представления_записи
13.2
спецификатор_длины :: = for атрибут use простое_выражение;
13.3
спецификатор_представления_перечисления :: = for простое_имя_типа use агрегат;
13.4
спецификатор_представления_записи :: =
    for простое_имя_типа use
    record [спецификатор_выравнивания]
      {спецификатор_компонента}
    end record;
спецификатор_выравнивания :: =
    at mod статическое_простое_выражение;
спецификатор_компонента :: =
    имя_компонента at статическое_простое_выражение
        range статический_диапазон;
13.5
спецификатор_адреса :: =
    for простое_имя use at простое_выражение;
13.8
оператор_кода :: = обозначение_типа ' агрегат_записи;
```

ИНФОРМАЦИОННЫЕ ДАННЫЕ

1. ИСПОЛНИТЕЛИ

А.А. Красилов, д-р. техн. наук, проф.; **В.М. Курочкин**, канд. техн. наук (руководители темы); **Ю.Н. Голубев**, канд. техн. наук; **В.И. Баранов**, канд. техн. наук; **В.Л. Лейтес**, канд. техн. наук; **В.А. Хитров**; **Н.Е. Богородская**; **В.П. Чепкасов**; **Е.П. Фадеева**; **А.П. Попов**; **В.В. Лукашев**, канд. техн. наук; **Н.Л. Жданова**; **В.Г. Коневских**; **Л.А. Андрианова**; **Т.С. Прокофьева**; **О.Е. Косырева**; **Е.Е. Полякова**; **Ю.П. Пузей**; **Е.В. Фадеева**; **А.Ф. Яблокова**; **Е.А. Обертынская**; **И.Ю. Гребенкина**; **О.Г. Кузьмина**; **Н.Г. Соболев**; **Н.М. Блохин**; **Н.К. Зубанова**

2. УТВЕРЖДЕН И ВВЕДЕН В ДЕЙСТВИЕ Постановлением Государственного комитета СССР по стандартам от 22.09.88 № 3217

3. ВВЕДЕН ВПЕРВЫЕ

4. Стандарт соответствует международному стандарту ИСО 8652–87

5. ССЫЛОЧНЫЕ НОРМАТИВНО-ТЕХНИЧЕСКИЕ ДОКУМЕНТЫ

Обозначение НТД, на который дана ссылка	Номер раздела, приложения
ГОСТ 27465–87	2.1, приложение 3

СОДЕРЖАНИЕ

1. Общие положения . . . 1
1.1. Область действия стандарта . . . 1
1.1.1. Содержание стандарта . . . 1
1.1.2. Согласованность реализации со стандартом . . . 2
1.2. Структура стандарта . . . 3
1.3. Цели и источники разработки . . . 3
1.4. Обзор свойств языка . . . 4
1.5. Метод описания и синтаксические обозначения . . . 8
1.6. Классификация ошибок . . . 9
2. Лексика . . . 10
2.1. Набор символов . . . 10
2.2. Лексемы, разделители и ограничители . . . 11
2.3. Идентификаторы . . . 13
2.4. Числовые литералы . . . 13
2.4.1. Десятичные литералы . . . 13
2.4.2. Литералы с основанием . . . 13
2.5. Символьные литералы . . . 14
2.6. Строковые литералы . . . 14
2.7 Комментарии . . . 15
2.8. Прагмы . . . 15
2.9. Зарезервированные слова . . . 16
2.10. Допустимые замены символов . . . 17
3. Описания и типы . . . 18
3.1. Описания . . . 18
3.2. Объекты и именованные числа . . . 19
3.2.1. Описания объектов . . . 21
3.2.2. Описание чисел . . . 22
3.3. Типы и подтипы . . . 23
3.3.1. Описания типов . . . 24
3.3.2. Описания подтипов . . . 25
3.3.3. Классификация операций . . . 26
3.4. Производные типы . . . 27
3.5. Скалярные типы . . . 29
3.5.1. Перечислимые типы . . . 30
3.5.2. Символьные типы . . . 31
3.5.3. Логические типы . . . 31
3.5.4. Целые типы . . . 31
3.5.5. Операции над дискретными типами . . . 33
3.5.6. Вещественные типы . . . 35
3.5.7. Плавающие типы . . . 36
3.5.8. Операции над плавающими типами . . . 38
3.5.9. Фиксированные типы . . . 40
3.5.10. Операции над фиксированными типами . . . 42
3.6. Индексируемые типы . . . 44
3.6.1. Ограничения индекса и дискретные диапазоны . . . 46
3.6.2. Операции над индексируемыми типами . . . 47
3.6.3. Строковый тип . . . 49
3.7. Именуемые типы . . . 49
3.7.1. Дискриминанты . . . 51

3.7.2. Ограничения дискриминантов . . . 52
3.7.3. Разделы вариантов . . . 54
3.7.4. Операции над именуемыми типами . . . 55
3.8. Ссылочные типы . . . 56
3.8.1. Неполные описания типов . . . 57
3.8.2. Операции над ссылочными типами . . . 58
3.9. Разделы описаний . . . 59
4. Имена и выражения . . . 60
4.1. Имена . . . 60
4.1.1. Индексируемые компоненты . . . 61
4.1.2. Отрезки . . . 61
4.1.3. Именуемые компоненты . . . 62
4.1.4. Атрибуты . . . 64
4.2. Литералы . . . 65
4.3. Агрегаты . . . 65
4.3.1. Агрегаты записей . . . 66
4.3.2. Агрегаты массивов . . . 67
4.4. Выражения . . . 69
4.5. Операции и вычисление выражения . . . 70
4.5.1. Логические операции и формы управления промежуточной проверкой . . . 71
4.5.2. Операции отношения и проверки принадлежности . . . 73
4.5.3. Бинарные аддитивные операции . . . 75
4.5.4. Унарные аддитивные операции . . . 76
4.5.5. Мультипликативные операции . . . 76
4.5.6. Операции высшего приоритета . . . 78
4.5.7. Точность операций с вещественными операндами . . . 79
4.6. Преобразование типа . . . 81
4.7. Квалифицированные выражения . . . 83
4.8. Генераторы . . . 84
4.9. Статические выражения и статические подтипы . . . 85
4.10. Универсальные выражения . . . 86
5. Операторы . . . 87
5.1. Простые и составные операторы. Последовательности операторов . . . 88
5.2. Операторы присваивания . . . 89
5.2.1. Присваивания массивов . . . 90
5.3. Условные операторы . . . 90
5.4. Операторы выбора . . . 91
5.5. Операторы цикла . . . 92
5.6. Операторы блока . . . 94
5.7. Операторы выхода . . . 95
5.8. Операторы возврата . . . 95
5.9. Операторы перехода . . . 96
6. Подпрограммы . . . 97
6.1. Описание подпрограммы . . . 97
6.2. Виды формальных параметров . . . 98
6.3. Тела подпрограмм . . . 100
6.3.1. Правила согласования . . . 101
6.3.2. Подстановка подпрограмм . . . 102
6.4. Вызовы подпрограмм . . . 102
6.4.1. Сопоставления параметров . . . 103
6.4.2. Опущенные параметры . . . 103
6.5. Функции . . . 105
6.6. Профиль типов параметров и результатов. Совмещение подпрограмм 105

6.7. Совмещение операций . . . 106
7. Пакеты . . . 107
7.1. Структура пакета . . . 107
7.2. Спецификации и описания пакетов . . . 108
7.3. Тела пакетов . . . 109
7.4. Описания личных типов и субконстант . . . 110
7.4.1. Личные типы . . . 111
7.4.2. Операции над личным типом . . . 112
7.4.3. Субконстанты . . . 114
7.4.4. Лимитируемые типы . . . 114
7.5. Пример пакета работы с таблицами . . . 116
7.6. Пример пакета обработки текстов . . . 117
8. Правила видимости . . . 119
8.1. Зона описания . . . 119
8.2. Области действия описаний . . . 120
8.3. Видимость . . . 121
8.4. Спецификаторы использования . . . 124
8.5. Описание переименования . . . 126
8.6. Стандартный пакет . . . 128
8.7. Контекст разрешения совмещения . . . 129
9. Задачи . . . 130
9.1. Спецификации задач и тела задач . . . 131
9.2. Задачные типы и задачные объекты . . . 133
9.3. Выполнение и активизация задачи . . . 133
9.4. Зависимость и завершение задач . . . 135
9.5. Входы, вызовы входов и операторы принятия . . . 137
9.6. Операторы задержки, длительность и время . . . 139
9.7. Операторы отбора . . . 141
9.7.1. Отбор с ожиданием . . . 141
9.7.2. Условные вызовы входов . . . 143
9.7.3. Временные вызовы входов . . . 144
9.8. Приоритеты . . . 145
9.9. Атрибуты задач и входов . . . 146
9.10. Операторы прекращения . . . 146
9.11. Разделяемые переменные . . . 147
9.12. Пример использования задачи . . . 148
10. Структура программы и результат компиляции . . . 149
10.1. Компилируемые модули. Библиотечные модули . . . 149
10.1.1. Спецификаторы контекста. Спецификаторы совместности . . . 151
10.1.2. Примеры компилируемых модулей . . . 152
10.2. Субмодули компилируемых модулей . . . 154
10.2.1. Примеры субмодулей . . . 155
10.3. Порядок компиляции . . . 157
10.4. Программная библиотека . . . 159
10.5. Предвыполнение библиотечных модулей . . . 159
10.6. Оптимизация программы . . . 160
11. Исключения . . . 160
11.1. Описания исключений . . . 161
11.2. Обработчики исключений . . . 162
11.3. Операторы возбуждения . . . 163
11.4. Обработка исключения . . . 163
11.4.1. Исключения, возбуждаемые при выполнении операторов . . . 163
11.4.2. Исключения, возбуждаемые при предвыполнении описаний . . . 166

11.5. Исключения, возбуждаемые при взаимодействии задач . . . 167
11.6. Исключения и оптимизация . . . 167
11.7. Подавление проверок . . . 169
12. Настраиваемые модули . . . 171
12.1. Описание настройки . . . 171
12.1.1. Формальные объекты настройки . . . 173
12.1.2. Формальные типы настройки . . . 174
12.1.3. Формальные подпрограммы настройки . . . 175
12.2. Настраиваемые тела . . . 176
12.3. Конкретизация настройки . . . 177
12.3.1. Правила сопоставления для формальных объектов . . . 180
12.3.2. Правила сопоставления для формальных личных типов . . . 180
12.3.3. Правила сопоставления для формальных скалярных типов . . . 181
12.3.4. Правила сопоставления для формальных индексируемых типов . . . 181
12.3.5. Правила сопоставления для формальных ссылочных типов . . . 182
12.3.6. Правила сопоставления для формальных подпрограмм . . . 183
12.4. Пример настраиваемого пакета . . . 184
13. Спецификаторы представления и особенности, зависящие от реализации . . . 185
13.1. Спецификаторы представления . . . 185
13.2. Спецификаторы длины . . . 187
13.3. Спецификаторы представления перечисления . . . 189
13.4. Спецификаторы представления записей . . . 189
13.5. Спецификаторы адреса . . . 191
13.5.1. Прерывания . . . 192
13.6. Изменение представления . . . 193
13.7. Системный пакет . . . 193
13.7.1. Зависящие от системы именованные числа . . . 194
13.7.2. Атрибуты представления . . . 195
13.7.3. Атрибуты представления вещественных типов . . . 196
13.8. Вставки машинных кодов . . . 197
13.9. Связь с другими языками . . . 198
13.10. Неконтролируемое программирование . . . 198
13.10.1. Неконтролируемое освобождение памяти . . . 198
13.10.2. Неконтролируемое преобразование типов . . . 199
14. Ввод-вывод . . . 199
14.1. Внешние файлы и файловые объекты . . . 200
14.2. Файлы последовательного и прямого доступа . . . 201
14.2.1. Управление файлами . . . 202
14.2.2. Последовательный ввод-вывод . . . 204
14.2.3. Спецификация пакета последовательного ввода-вывода . . . 204
14.2.4. Прямой ввод-вывод . . . 205
14.2.5. Спецификация пакета прямого ввода-вывода . . . 206
14.3. Ввод-вывод текстов . . . 207
14.3.1. Управление файлами . . . 209
14.3.2. Файлы ввода и вывода по умолчанию . . . 209
14.3.3. Спецификации длин строчек и страниц . . . 210
14.3.4. Операции над столбцами, строчками и страницами . . . 211
14.3.5. Процедуры обмена . . . 214
14.3.6. Ввод-вывод символов и строк . . . 216
14.3.7. Ввод-вывод для целых типов . . . 217
14.3.8. Ввод-вывод для вещественных типов . . . 219
14.3.9. Ввод-вывод для перечислимых типов . . . 221
14.3.10. Спецификация пакета ввода-вывода текста . . . 223
14.4. Исключения при вводе-выводе . . . 226

14.5. Спецификация пакета исключений ввода-вывода 227
14.6. Ввод-вывод низкого уровня 227
14.7. Пример ввода-вывода 228
Приложение 1. Атрибуты, предопределенные в языке 230
Приложение 2. Прагмы, предопределенные в языке 234
Приложение 3. Предопределенное окружение языка 236
Приложение 4. Характеристики, зависящие от реализации 242
Приложение 5. Термины и определения 243
Приложение 6. Сводка синтаксиса 248
Информационные данные 258

Редактор *В.П. Огурцов*
Технический редактор *Е.В. Минакова*
Корректор *Н.Л. Шнайдер*

Сдано в наб. 24.10.88 Подп. в печ. 16.03.89 16,5 усл. печ. л. 16,63 усл. кр.-отт. 22,24 уч.-изд. л. Тираж 16000 Цена 1 р. 20 к.

Ордена „Знак Почета" Издательство стандартов, 123840, Москва, ГСП, Новопресненский пер., 3

Набрано в Издательстве стандартов на НПУ

Вильнюсская типография Издательства стандартов, ул. Даряус и Гирено, 39. Зак. 857

gost.e-ecolog.ru