1. Элементарные понятия.

1.1 "Сюрпризы" переводной терминологии

Прежде чем приступить к непосредственному знакомству с Адой, есть необходимость заметить, что в англоязычной литературе, посвященной вычислительной технике в целом и программированию в частности, достаточно часто встречаются такие английские термины как *operator*, *operation* и *statement*. При сложившейся практике перевода такой документации на русский язык, эти термины, как правило, переводят следующим образом:

operation	операция
operator	операция, оператор
statement	оператор

На первый взгляд, проблема перевода данных терминов достаточно безобидна. Однако, при таком традиционном подходе, добившись приемлемой благозвучности изложения материала, достаточно сложно избежать неоднозначностей и даже путаницы при обсуждении некоторых основополагающих вопросов. Также стоит особо отметить, что стандарт языка Ада строго различает такие понятия.

В действительности, смысловое значение данных терминов, при их употреблении в англоязычной документации, как правило, имеет следующее русскоязычное толкование:

operation	непосредственно обозначает понятие операции, функции или какого-либо действия, в совокупности составляющего выполнение команды (или набора команд) процессора
operator	обозначает понятие знака или обозначения операции, функции или какого-либо действия, что подразумевает не столько само действие для выполнения операции, сколько обозначение операции в тексте программы
statement	элемент текста программы, выражающий целостное законченное действие (или набор действий)

Таким образом, исходя из всего выше сказанного, в данной работе, для достижения однозначности, принято следующее соответствие терминов:

operation	операция
operator	знак операции
statement	инструкция

Хотя такое решение выглядит несколько не привычно, оно не должно вызвать трудности при рассмотрении материала представленного в этой работе.

1.2 Первая программа

Для того, чтобы дать "почувствовать", что представляет из себя программа написанная на языке Ада рассмотрим простую программу. Традиционно, первая программа - это программа которая выводит на экран приветствие: "Hello World!". Не будем нарушать традицию. Итак, на Аде такая программа будет иметь следующий вид:

```
with Ada.Text_IO;
use Ada.Text_IO;
```

```
procedure Hello is
begin
    Put_Line("Hello World!");
end Hello;
```

Давайте детально рассмотрим из каких частей состоит текст этой программы. Строка "**procedure** Hello **is**" является заголовком процедуры и она указывает имя нашей процедуры. Далее, между зарезервированными словами **begin** и **end**, располагается тело процедуры Hello. В этом примере тело процедуры очень простое и состоит из единственной инструкции "Put_Line("Hello World!");". Эта инструкция осуществляет вывод приветствия на экран, вызывая процедуру Put_Line. Процедура Put_Line располагается в пакете текстового ввода/вывода *Ada.Text_IO*, и становится доступной благодаря спецификации контекста в инструкциях "**with** Ada.Text_IO;" и "**use** Ada.Text_IO;" (спецификация контекста необходима для указания используемых библиотечных модулей). Здесь, спецификатор контекста состоит из двух спецификаторов: спецификатора совместности **with** и спецификатора использования **use**. Спецификатор совместности **with** указывает компоненты которые будут использоваться в данном компилируемом модуле. Спецификатор использования **use** делает имена используемых объектов непосредственно доступными в данном компилируемом модуле.

Программа Hello настолько проста, что в ней нет ни переменных, ни какой-либо обработки данных, поэтому, несколько забегая вперед, приведем общий вид процедуры.

```
with ... ;
use  ... ;                                  спецификаторы контекста, указывающие используемые модули (могут отсутствовать)

procedure < имя процедуры > ... is
                                            спецификация процедуры, определяющая имя процедуры и ее параметры (если они есть)

      . . .                                 описательная (или декларативная) часть, которая может содержать описания типов, переменных,
                                            констант и подпрограмм

begin

      . . .                                 исполняемая часть процедуры, которая описывает алгоритм работы процедуры

end < имя процедуры >;
                                            здесь, указание имени процедуры не является обязательным
```

Необходимо заметить, что в отличие от языков C/C++, которые имеет функцию **main**, и языка Паскаль, который имеет **program**, в Аде, любая процедура без параметров может быть подпрограммой main (другими словами - головной программой). Таким образом, процедура без параметров может быть выбрана как головная программа во время линковки.

Теперь, приведем еще один простой пример, в котором, для выдачи сообщения приветствия, используется ранее рассмотренная процедура `Hello`:

```
with Hello;    -- указывает на использование показанной ранее
               -- процедуры Hello

procedure Use_Hello is
begin
   Hello;      -- вызов процедуры Hello
end Use_Hello;
```

1.3 Библиотека и компилируемые модули

В общем случае, программа на языке Ада представляет собой один или несколько программных модулей, которые могут компилироваться как совместно, так и раздельно. Кроме того, программные модули являются основой построения библиотек Ады, поэтому их также называют библиотечными модулями. Программные модули бывают четырех видов:

- **Подпрограммы** - Являются основным средством описания алгоритмов. Различают два вида подпрограмм: процедуры и функции. Процедура - это логический аналог некоторой именованной последовательности действий. Функция - логический аналог математической функции - используется для вычисления какого-либо значения.

- **Пакет** - Основное средство для определения набора логически взаимосвязанных понятий. В простейшем случае в пакете специфицируются описания типов и общих объектов. В более общем случае в нем могут специфицироваться группы взаимосвязанных понятий, включающих подпрограммы, причем, некоторые описываемые в пакете сущности могут быть "скрыты" от пользователя, что дает возможность предоставления доступа только к тем ресурсам пакета, которые необходимы пользователю и, следовательно, должны быть для него доступны.

- **Задача** или **задачный модуль** - Средство для описания последовательности действий, причем, при наличии нескольких таких последовательностей они могут выполняться параллельно. Задачи могут быть реализованы на многомашинной или многопроцессорной вычислительной конфигурации, либо на единственном процессоре в режиме разделения времени. Синхронизация достигается путем обращения ко входам, которые подобно подпрограммам могут иметь параметры, с помощью которых осуществляется передача данных между задачами.

- **Настраиваемые модули** - Средство для параметризации подпрограмм или пакетов. В ряде случаев возникает необходимость обрабатывать объекты, которые отличаются друг от друга количеством данных, типами или какими-либо другими количественными или качественными характеристиками. Если все эти изменяемые характеристики вынести из подпрограммы или пакета, то получится некоторая заготовка (или шаблон), которую можно настроить на конкретное выполнение. Непосредственно выполнить настраиваемый модуль

нельзя. Но из него можно получить экземпляр настроенного модуля (подпрограмму или пакет), который пригоден для выполнения.

Каждый программный модуль обычно состоит из двух частей: спецификации и тела. Спецификация описывает интерфейс к модулю, а тело - его реализацию. Примечательно, что спецификация и тело программного модуля являются самостоятельными компилируемыми модулями, то есть, они могут компилироваться раздельно. Разбиение модуля на спецификацию и тело, а также возможность раздельной компиляции позволяют разрабатывать, кодировать и тестировать любую программу или систему как набор достаточно независимых компонентов. Такой подход полезен при разработке больших программных систем.

1.4 Лексические соглашения

Лексические соглашения описывают допустимые символы и последовательности символов, которые используются при обозначении идентификаторов, определяющих имена переменных и констант, подпрограмм и пакетов, указывают правила написания числовых значений, а также описывают некоторые специальные последовательности символов, используемые языком программирования. Согласно требований стандарта, реализация должна поддерживать как минимум 200-символьные строки исходного текста (максимальная длина лексических элементов - не определена).

1.4.1 Комментарии

Начнем с комментариев. Для облегчения понимания алгоритма работы программы, в текст программы могут, и должны помещаться комментарии. Комментарий начинается с двух символов дефиса "--" и продолжается до конца строки. Пример:

```
-- это комментарий
  --- это тоже комментарий
```

1.4.2 Идентификаторы

Теоретически, согласно требований стандарта, идентификаторы могут быть любой длины, однако, длина может быть ограничена реализацией конкретного компилятора. Общие правила таковы:

1. Идентификатор может состоять из букв, цифр и символов подчеркивания.
2. Идентификатор **обязан** начинаться с символа.
3. В идентификаторе **нельзя** использовать несколько символов подчеркивания подряд.
4. Символ подчеркивания **не** может быть первым и последним символом в идентификаторе.
5. Все идентификаторы в ADA **не** зависят от регистра символов.

Например:

```
Apple, apple, APPLE     -- один и тот же идентификатор
Max_Velocity_Attained
```

```
Minor_Number_           -- недопустимо, завершающий символ - подчеркивание
Minor__Revision         -- недопустимо, последовательность подчеркиваний
```

1.4.3 Литералы

Литералы служат для явного указания значения некоторого типа, сохраняемого в программе. Различают числовые, символьные и строковые литералы.

Числовые литералы, как не трудно догадаться, используются для представления численных значений. Они могут содержать в себе символы подчеркивания (для удобочитаемости), однако, они **не** могут начинаться или заканчиваться символом подчеркивания, или содержать более одного символа подчеркивания подряд. Различают числовые литералы для представления целочисленных и вещественных значений.

Примеры целочисленных литералов, представляющих значение числа 2000:

```
2000
2_000
2E3     -- для целочисленных литералов разрешена экспоненциальная форма
  2E+3
```

Возможно представление чисел в разных системах счисления, например, представление десятичного числа 12 может быть задано следующими литералами:

```
2#1100#   -- двоичная система счисления
8#14#     -- восьмеричная
10#12#    -- десятичная (здесь, указана явно)
16#C#     -- шестнадцатиричная
  7#15#       -- семиричная
```

Литералы, описывающие вещественные значения, содержат точку и обязаны иметь хотя бы по одной цифре до и после точки. Примеры:

```
3.14
100.0
  0.0
```

Символьные литералы обозначают одиночные символы, и для их обозначения используются одинарные кавычки. Например:

```
'a'
  'b'
```

Примечание:
В отличие от языка Паскаль, в Аде **нельзя** производить присваивание символьного литерала строковой переменной (подробности о работе со строками в Аде мы пока отложим на потом).

Строковые литералы предназначены для обозначения строковых значений, и для их обозначения используются двойные кавычки. Например:

```
"это строковый литерал"
  "a"                     -- это тоже строковый литерал, хотя и односимвольный
```

Примечание:
Строковый литерал не может непосредственно содержать символ табуляции, хотя значение строковой переменной или константы - может. Это достигается путем конкатенации строки и символа, или вставкой символа в строку.

1.4.4 Зарезервированные слова

Некоторые слова, такие как **`with`**, **`procedure`**, **`is`**, **`begin`**, **`end`** и т.д., являются частью самого языка программирования. Такие слова называют зарезервированными (или ключевыми) и они не могут быть использованы в программе в качестве имен идентификаторов. Полный список зарезервированных слов Ады приводится ниже:

```
  abort       else          new           return
  abs         elsif         not           reverse
* abstract    end           null
  accept      entry         select
  access      exception     separate
* aliased     exit          of            subtype
  all         or
  and         for           others        * tagged
  array       function      out           task
  at                        terminate
              generic       package       then
  begin       goto          pragma        type
  body        private
              if            procedure
  case        in            * protected   * until
  constant    is            use
                            raise
  declare     range         when
  delay       limited       record        while
  delta       loop          record        while
  digits      renames
    do        mod           * requeue     xor
```

Примечание:
Зарезервированные слова помеченные звездочкой введены стандартом Ada95.

1.5 Методы Ады: подпрограммы, операции и знаки операций

Методами Ады являются подпрограммы (процедуры и функции), а также операции и знаки операций (возможно более корректно будет звучать: с помощью подпрограмм осуществляется реализация действий, выполняемых операциями и знаками

операций). Необходимо отметить, что стандарт Ады строго различает понятия знаков операций (*operators*) и операций (*operations*).

Знаки операций представляются следующими символами (или комбинациями символов): "**=**", "**/=**", "**<**", "**>**", "**<=**", "**>=**", "**&**", "**+**", "**-**", "**/**", "*****". Другие знаки операций выражаются зарезервированными словами: "**and**", "**or**", "**xor**", "**not**", "**abs**", "**rem**", "**mod**", - или могут состоят из нескольких зарезервированных слов: "**and then**", "**or else**". Ада позволяет осуществлять программисту совмещение (*overloading*) знаков операций (в современной литературе по Си++ это часто называется как "перегрузка операторов").

> В общем случае, **совмещением** (*overloading*) называют механизм, который позволяет различным сущностям использовать одинаковые имена.

Использование "**use type**" делает знаки операций именованных типов локально видимыми. Кроме того, их можно сделать локально видимыми используя локальное переименование.

Операции включают в себя присваивание, проверку принадлежности диапазону и любые другие именованные операции. Операции, также как и знаки операций, допускают совмещение.

Следует заметить, что Ада накладывает некоторые ограничения на использование совмещений: совмещения не допускаются для операций присваивания и проверки принадлежности диапазону, а также для знаков операций "**and then**" и "**or else**".

Операция присваивания обозначается комбинацией символов "**:=**". Она предопределена для всех нелимитированных типов. Операция присваивания не может быть совмещена или переименована. Присваивание запрещено для лимитированных типов. Необходимо подчеркнуть, что операция присваивания в Аде, в отличие от языков C/C++, не возвращает значение и не обладает побочными эффектами.

Еще одной разновидностью операций является операция проверки принадлежности диапазону, которая обозначается с помощью зарезервированного слова "**in**". Для выполнения проверки на не принадлежность "**in**" может комбинироваться с "**not**" - "**not in**". Проверка на принадлежность диапазону разрешена для всех типов Ады, включая лимитированные.

Другие операции могут быть описаны программистом. Как правило, описания таких операций выполняются в спецификации пакета, а реализация операций выполняется с помощью соответствующих подпрограмм.

1.6 Инструкции, выражения и элаборация

Очевидно, что исполнение инструкций осуществляется во время выполнения программы с целью выполнить какие-либо действия. Также, во время выполнения программы осуществляются вычисления различных выражений для получения

значений каких-либо типов. Кроме того, во время выполнения программы происходит вычисление различных имен, которые указывают на соответствующие объекты (содержащие какие-либо значения) или другие сущности (такие как подпрограммы и типы).

Некоторые конструкции языка содержат описательные части, сопровождаемые последовательностями инструкций. Например, тело процедуры может иметь следующий вид:

```
procedure P( ... ) is
   I: Integer := 1;   -- описательная часть
   ...
begin
   ...          -- последовательность инструкций
   I := I * 2;
   ...
end P;
```

Перед выполнением тела процедуры происходит элаборация (*elaboration*) всех описаний, которые указаны в описательной части. Последовательность элаборации описаний определяется порядком их следования в описательной части. Эффект от элаборации описаний заключается в создании сущностей, определенных в описаниях, и в выполнении прочих действий, которые специфичны для описаний. Например, элаборация описания переменной может осуществить инициализацию этой переменной значением, которое определяется результатом вычисления какого-либо выражения. Достаточно часто значения подобных выражений могут быть вычислены в процессе компиляции программы.

После завершения элаборации, осуществляется исполнение последовательности инструкций, в порядке их следования (за исключением случаев, когда осуществляется передача управления в какое-либо другое место, отличное от последующей инструкции). Инструкция присваивания позволяет заменить значение переменной результатом вычисления выражения того же самого типа. Обычно, присваивание осуществляется простым побитовым копированием значения, которое получено в результате вычисления выражения. Однако, в случае нелимитированных контролируемых типов, после осуществления побитового копирования, пользователь (при необходимости) может определить дополнительную последовательность действий. Инструкции **`case`** и **`if`** позволяют осуществлять выбор выполнения определенной последовательности инструкций, полагаясь на результат вычисления какого-нибудь выражения. Инструкция **`loop`** позволяет повторять выполнение последовательности каких-либо инструкций, согласно выбранной схемы итерации, или до обнаружения инструкции **`exit`**. Инструкция **`goto`** осуществляет передачу управления в место отмеченное соответствующей меткой.

Выражения могут быть использованы в различном контексте, как в описаниях, так и в инструкциях. Используемые в языке Ада выражения, во многом подобны выражениям, которые используются в большинстве современных языков программирования. Они могут содержать обращения к переменным, константам и литералам, кроме того они могут использовать любые операции, которые возвращают

значения. Результатом вычисления любого выражения является значение. Каждое выражение имеет определенный тип, который известен на этапе компиляции.

Во многих случаях результаты вычисления выражений и ограничения подтипов определяются статически (существует возможность отключения некоторых динамических проверок ограничений подтипов с помощью использования соответствующих опций компилятора). Более того, достаточно часто компилятор Ады требует чтобы вычисление некоторых выражений и подтипов осуществлялось на этапе компиляции программы. Например, в общем случае вся информация об описании известна во время компиляции, следовательно, элаборация во время выполнения программы не потребует выполнения какого-либо машинного кода. Язык определяет механизмы согласно которых Ада-компиляторы могут осуществлять предварительную элаборацию некоторых модулей, то есть, реальные действия, которые необходимы для осуществления элаборации, выполняются однократно, на этапе компиляции программы, вместо того, чтобы выполнять их при каждом запуске программы.

1.7 Директивы компилятора

Бывают случаи, когда в исходном тексте необходимо указывать какую-либо дополнительную информацию, которая предназначена сугубо для компилятора. Например, такая информация может предоставлять компилятору дополнительные сведения о режимах трансляции программного модуля, который компилируется в текущий момент времени (оптимизация генерируемого двоичного кода, вставка отладочного кода и т.д.) или управлять распечаткой листинга трансляции.

Для Ады, как и для многих других современных языков и сред программирования, такими средствами передачи дополнительной информации компилятору являются директивы компилятора. При указании директивы компилятору Ада использует зарезервированное слово **`pragma`**. Общий вид указания директивы компилятору следующий:

```
pragma < имя_директивы > ( < параметры_директивы > );
```

Стандарт языка Ада определяет 39 директив, список которых представлен в приложении L (*Annex L*) руководства по языку программирования Ада (*RM-95*). Кроме того, конкретная реализация компилятора может обеспечивать дополнительные директивы компилятора которые, как правило, описываются в сопроводительной документации компилятора.

Выражаясь не строго, можно заметить, что директивы компилятора не изменяют общий смысл программы.

2. Скалярные типы данных языка Ада.

К скалярным типам относятся типы данных которые определяют соответствующие упорядоченные множества значений. Эта глава описывает скалярные типы данных Ады, а также атрибуты и операции допустимые для этих типов.

Предопределенный пакет *Standard* содержит описания стандартных типов, таких как `Integer`, `Float`, `Boolean`, `Character` и `Wide_Character`, а также определяет операции, которые допускается производить над этими типами.

Следующие знаки операций допустимы для всех скалярных типов:

```
=, /=             проверка на равенство/не равенство
<, <=, >, >=      меньше, меньше или равно, больше, больше или равно
in, not in        проверка принадлежности к диапазону
```

Перед тем как приступить к непосредственному детальному обсуждению скалярных типов Ады, необходимо сделать некоторое общее введение в систему типов языка Ада

2.1 Введение в систему типов языка Ада

Данные - это то, что обрабатывает программа. Практически, любой современный язык программирования определяет свои соглашения и механизмы для разделения данных на разные типы. Известно, что Ада - это язык со строгой типизацией. И хотя это не единственное свойство Ады, но, пожалуй, это свойство наиболее широко известно.

Понятие типа данных Ады подразумевает, что:

1. каждый тип данных имеет свое имя
2. каждый тип данных имеет свое множество допустимых значений
3. для каждого типа данных определено свое множество допустимых операций и знаков операций
4. строгое разделение объектов одного типа данных от объектов любого другого типа данных

Исходя из сказанного, в отличие от других языков программирования, Ада **не допускает** прямого присваивания значений одного типа данных другому типу данных и/или любого **неявного** преобразования значений одного типа в значения другого типа. В случае необходимости преобразования значений одного типа в значения другого типа можно указать (или описать) требуемое правило преобразования, но такое правило преобразования должно быть указано явно.

На первый взгляд, строгое требование явного указания или описания всех правил преобразования выглядит раздражающе (особенно для тех кто привык к C) - без явного указания таких правил компилятор просто отказывается компилировать вашу программу. Однако, в последствии это предохраняет от долгих и мучительных поисков, в процессе отладки, тех "сюрпризов" неявного преобразования и/или некорректного присваивания значений различных типов, которые допускаются в других языках программирования. Таким образом, подобный подход является одним из основополагающих факторов обеспечения надежности программного обеспечения.

Ада обладает достаточным набором предопределенных встроенных типов, а также предоставляет богатые возможности для описания новых типов данных и их подтипов (*subtype*). Кроме того, в языке представлены средства, которые позволяют гибко управлять внутренним представлением вновь создаваемых типов данных.

Приводимая ниже диаграмма демонстрирует общую организацию системы типов Ады.

```
Все типы
  |
  |- Простые типы
  |    |
```

```
|    |- Скалярные типы
|    |    |
|    |    |- Вещественные (Real)
|    |    |    |
|    |    |    |- Универсальный Вещественный (Universal_Real)    -- все вещественные
|    |    |    |                                                 -- литералы
|    |    |    |
|    |    |    |- Корневой Вещественный       (Root_Real)        -- только Ada95
|    |    |         |
|    |    |         |- с плавающей точкой      (Floating Point)
|    |    |         |- с фиксированной точкой (Fixed Point)
|    |    |              |
|    |    |              |- с обычной фиксированной точкой
|    |    |              |      (Ordinary Fixed Point)
|    |    |              |
|    |    |              |- с десятичной фиксированной точкой   -- только Ada95
|    |    |                     (Decimal Fixed Point)
|    |    |
|    |    |- Дискретные типы
|    |         |
|    |         |- Целые типы
|    |         |     |
|    |         |     |- Универсальный Целый (Universal_Integer)  -- все
|    |         |     |                                           -- целочисленные
|    |         |     |                                           -- литералы
|    |         |     |
|    |         |     |- Корневой Целый (Root_Integer)            -- только Ada95
|    |         |         |
|    |         |         |- Знаковые Целые
|    |         |         |- Модульные Целые (Modular Integer)    -- только Ada95
|    |         |
|    |         |- Перечислимые
|    |              |
|    |              |- Символьные   (Character, Wide_Character)
|    |              |- Логический   (Boolean)
|    |              |- Определяемые пользователем
|    |
|    |-- Ссылочные типы / указатели   (Access)
|         |
|         |- Ссылки на объекты
|         |- Ссылки на подпрограммы                          -- только Ada95
|
|- Составные типы
       |
       |-- Массивы (Array)
       |     |
       |     |- Строки  (String)
       |     |- Другие, определяемые пользователем массивы
       |
       |-- Записи  (Record)
       |
       |-- Тэговые Записи (Tagged Record)                     -- только Ada95
       |
       |-- Задачи  (Task)
       |
       |-- Защищенные типы  (Protected)                       -- только Ada95
```

Примечательно, что, в отличие от языка Паскаль, Ада не предоставляет предопределенного

механизма поддержки множественных типов. Однако, богатые средства описания типов Ады, при необходимости, позволяют программисту конструировать подобные типы без значительных усилий.

2.2 Целочисленные типы

2.2.1 Предопределенный тип `Integer`

Предопределенный целочисленный тип `Integer` описан в пакете *Standard* (пакет *Standard* не нужно указывать в инструкциях спецификации контекста **`with`** и **`use`**). Точный диапазон целочисленных значений, предоставляемых этим типом, зависит от конкретной реализации компилятора и/или оборудования. Однако, стандарт определяет минимально допустимый диапазон значений для этого типа от -(2 ** 15) до +(2 ** 15 - 1) (например, в случае 32-битных систем, таких как *Windows* или *Linux*, для реализации компилятора GNAT диапазон значений типа `Integer` будет от -(2 ** 31) до +(2 ** 31 - 1)).

Кроме типа `Integer`, в пакете *Standard* предопределены два его подтипа (понятие подтипа будет рассмотрено позже) `Natural` и `Positive`, которые, соответственно, описывают множества не отрицательных (натуральных) и положительных целых чисел.

2.2.2 Тип `Universal_Integer`

Для предотвращения необходимости явного преобразования типов при описании целочисленных констант, Ада предлагает понятие универсального целого типа - `Universal_Integer`. Все целочисленные литералы принадлежат типу `Universal_Integer`. Многие атрибуты языка (обсуждаемые позже) также возвращают значения универсального целого типа. Установлено, что тип `Universal_Integer` совместим с любым другим целочисленным типом, поэтому, в арифметических выражениях, компилятор будет автоматически преобразовывать значения универсального целого типа в значения соответствующего целочисленного типа.

2.2.3 Описание целочисленных констант

При хорошем стиле программирования, принято присваивать целочисленным значениям символьные имена. Это способствует улучшению читабельности программы и, при необходимости, позволяет легко вносить в нее изменения.

Приведем простой пример описания константы (именованного числа):

```
Max_Width : constant := 10_000; -- 10_000 - имеет тип Universal_Integer
                                        -- это не тип Integer
```

2.2.4 Тип `Root_Integer`

Модель целочисленной арифметики Ады базируется на понятии неявного типа `Root_Integer`. Этот тип используется как базовый тип для всех целочисленных типов Ады. Другими словами - все целочисленные типы являются производными от типа `Root_Integer` (см. Производные типы). Диапазон значений типа `Root_Integer` определяется как `System.Min_Int..System.Max_Int`. Все знаки арифметических операций описаны так, чтобы они могли выполняться над этим типом.

При описании нового целочисленного типа возможны два различных подхода, которые можно проиллюстрировать на следующем примере:

```
type X is new Integer range 0 .. 100;
type Y is range 0 .. 100;
```

Здесь, тип `X` описывается как производный от типа `Integer` с допустимым диапазоном значений от 0 до 100. Исходя из этого, для типа `X` базовым типом будет тип `Integer`.

Тип `Y` описывается как тип с допустимым диапазоном значений от 0 до 100, и при его описании не указан тип-предок. В таком случае, он будет производным от типа `Root_Integer`, но его базовый диапазон не обязательно должен быть таким же как у `Root_Integer`. В результате, некоторые системы могут размещать экземпляры объектов такого типа и его базового типа в одном байте. Другими словами, определение размера распределяемого места под объекты такого типа возлагается на компилятор.

2.2.5 Примеры целочисленных описаний

Ниже приводятся примеры различных целочисленных описаний Ады.

```
-- описания целочисленных статических переменных


Count           : Integer;
X, Y, Z         : Integer;
Amount          : Integer := 0;



-- описания целочисленных констант (иначе - именованных чисел)


Unity           : constant Integer := 1;
Speed_Of_Light  : constant := 300_000; -- тип Universal_Integer
A_Month         : Integer range 1..12;



-- описания целочисленных типов и подтипов
-- ( см. разделы "Подтипы" и "Производные типы" )


subtype Months is Integer range 1..12;        -- огранниченный тип Integer
-- подтипы - совместимы с их базовым типом (здесь - Integer)
-- например, переменная типа Month может быть "смешана" с переменными
-- типа Integer


type File_Id is new Integer; -- новый целочисленный тип, производный
                             -- от типа Integer


type Result_Range is new Integer range 1..20_000;
-- производный тип с объявлением ограничения


type Other_Result_Range is range 1..100_000;
-- тип производный от Root_Integer
-- при этом, компилятор будет выбирать подходящий размер целочисленного значения
-- для удовлетворения требований задаваемого диапазона
```

2.2.6 Предопределенные знаки операций для целочисленных типов

Следующие знаки операций предопределены для каждого целочисленного типа:

```
+, -                унарные плюс и минус
+, -, *, /          сложить, вычесть, умножить и разделить
**                  возведение в степень (только целые значения степени)
mod                 модуль
rem                 остаток
```

```
abs                абсолютное значение
```

2.2.7 Модульные типы

Все целочисленные типы, которые мы рассматривали ранее, известны как целые числа со знаком. Для таких типов соблюдается правило - если в случае вычисления результат выходит за диапазон допустимых значений типа, то генерируется ошибка переполнения. Такие целочисленные типы были представлены стандартом Ada83.

Стандарт Ada95 разделяет целочисленные типы на целые числа со знаком и модульные типы. По существу, модульные типы являются целыми числами без знака. Характерной особенностью таких типов является свойство цикличности арифметических операций. Таким образом, модульные типы соответствуют целочисленным беззнаковым типам в других языках программирования (например: `Byte`, `Word`... - в реализациях Паскаля; `unsigned_short`, `unsigned`... - в C/C++).

В качестве простого примера рассмотрим следующий фрагмент кода:

```
   ...
type  Byte  is mod 2 ** 8;     -- (2 ** 8) = 256
   Count : Byte := 255;
begin
   Count := Count + 1;
      . . .
```

Здесь не производится генерация ошибки в результате выполнения сложения. Вместо этого, переменная `Count`, после выполнения сложения, будет содержать 0.

Кроме этого, с модульными типами удобно использовать знаки битовых операций "**`and`**", "**`or`**", "**`xor`**" и "**`not`**". Такие операции трактуют значения модульного типа как битовый шаблон. Например:

```
type  Byte  is mod 2 ** 8;     -- (2 ** 8) = 256
   Some_Var_1  : Byte;
   Some_Var_2  : Byte;
   Mask        : constant := 16#0F#
begin
   ...
   Some_Var_2 := Some_Var_1 and Mask;
      . . .
```

Поскольку модульные типы не имеют отрицательных значений, для них допускается смешивание знаков битовых операций со знаками арифметических операций в одном выражении.

Следует заметить, что хотя при описании модульных типов зачастую используют степень двойки, использование двоичной системы счисления при описании модульных типов не является обязательным требованием.

Ада допускает выполнение преобразований беззнаковых чисел модульных типов в числа со знаком и обратно. При этом, производится проверка результата преобразования на допустимость диапазону значений типа назначения. В случае неудачи будет сгенерировано исключение *Constraint_Error*. Например:

```
type  Unsigned_Byte  is mod 2 ** 8;     -- (2 ** 8) = 256
```

```
type  Signed_Byte    is range -128 .. +127;


U : Unsigned_Byte := 150;
S : Signed_Byte   := Signed_Byte(U);  -- здесь будет сгенерировано исключение
                                                    --   Constraint_Error
```

Этот код будет вызывать генерацию исключения *Constraint_Error*.

В следующем параграфе приводятся описания предопределенных модульных типов представленных в пакете *Interfaces*. Заметим, что для этих модульных типов (они описаны с использованием двоичной системы счисления) предопределены операции побитного сдвига и вращения.

2.2.8 Дополнительные целочисленные типы системы компилятора *GNAT*

Стандарт языка Ада допускает определять в реализации Ада-системы собственные дополнительные целочисленные типы. Таким образом, в пакете *Standard* системы компилятора *GNAT* определены дополнительные целочисленные типы:

```
type   Short_Short_Integer is range -(2 ** 7) .. +(2 ** 7 - 1);
type   Short_Integer       is range -(2 ** 15) .. +(2 ** 15 - 1);
type   Long_Integer        is range -(2 ** 31) .. +(2 ** 31 - 1);
type   Long_Long_Integer   is range -(2 ** 63) .. +(2 ** 63 - 1);
```

Кроме этого, стандарт требует наличия определения дополнительных 8-, 16-, 32- и 64-битных целочисленных типов в пакете *Interfaces*:

```
type   Integer_8     is range   -2 **  7 .. 2 **  7 - 1;
type   Integer_16    is range   -2 ** 15 .. 2 ** 15 - 1;
type   Integer_32    is range   -2 ** 31 .. 2 ** 31 - 1;
type   Integer_64    is range   -2 ** 63 .. 2 ** 63 - 1;


type   Unsigned_8    is mod   2 **  8;
type   Unsigned_16   is mod   2 ** 16;
type   Unsigned_32   is mod   2 ** 32;
type   Unsigned_64   is mod   2 ** 64;
```

2.3 Вещественные типы

Ада предусматривает два способа представления вещественных чисел: представление вещественных величин с плавающей точкой и представление вещественных величин с фиксированной точкой. Кроме этого вы можете использовать типы вещественных величин с десятичной фиксированной точкой.

2.3.1 Вещественные типы с плавающей точкой, тип `Float`

Вещественные типы с плавающей точкой имеют неограниченный диапазон значений и точность, определяемую количеством десятичных цифр после запятой. Представление чисел с плавающей точкой имеет фиксированную относительную погрешность.

Пакет *Standard* предоставляет предопределенный вещественный тип с плавающей точкой `Float`, который обеспечивает точность в шесть десятичных цифр после запятой:

```
type  Float  is digits 6 range -16#0.FFFF_FF#E+32 .. 16#0.FFFF_FF#E+32;
                                  --    -3.40282E+38 ..  3.40282E+38
```

В пакете *Standard* компилятора *GNAT*, для 32-битных систем Linux и Windows, дополнительно представлены еще несколько вещественных типов с плавающей точкой (фактические значения

констант для различных платформ отличаются):

```
type  Short_Float     is digits 6
  range -16#0.FFFF_FF#E+32 .. 16#0.FFFF_FF#E+32;
  --  -3.40282E+38 ..  3.40282E+38

type  Long_Float      is digits 15
  range -16#0.FFFF_FFFF_FFFF_F8#E+256 .. 16#0.FFFF_FFFF_FFFF_F8#E+256;
  --  -1.79769313486232E+308 ..  1.79769313486232E+308

type  Long_Long_Float is digits 18
  range -16#0.FFFF_FFFF_FFFF_FFFF#E+4096 .. 16#0.FFFF_FFFF_FFFF_FFFF#E+4096;
    --    -1.18973149535723177E+4932 ..  1.18973149535723177E+4932
```

Ниже следуют примеры описаний вещественных величин с плавающей точкой.

```
X         : Float;
A, B, C   : Float;
Pi        : constant Float := 3.14_2;
Avogadro  : constant := 6.027E23;        -- тип Universal_Float


subtype Temperatures is Float range 0.0..100.0;
type Result is new Float range 0.0..20_000.0;


type Velocity is new Float;
type Height is new Float;
-- нельзя случайно смешивать Velocity и Height
-- без явного преобразования типов.


type Time is digits 6 range 0.0..10_000.0;
-- в этом диапазоне требуемая точность - шесть десятичных цифр
-- после точки


type Degrees is digits 2 range -20.00..100.00;
-- требуемая точность - две десятичных цифры после точки
```

Следующие знаки операций предопределены для каждого вещественного типа с плавающей точкой.

```
+, -, *, /
**                возведение в степень (только целые значения степени)
abs               абсолютное значение
```

2.3.2 Вещественные типы с фиксированной точкой, тип `Duration`

Представление чисел с фиксированной точкой имеет более ограниченный диапазон значений и указанную абсолютную погрешность, которая задается как **`delta`** этого типа.

В пакете *Standard* предоставлен предопределенный вещественный тип с фиксированной точкой `Duration`, который используется для представления времени и обеспечивает точность измерения времени в 50 микросекунд:

```
type Duration is delta 0.000000001
    range -((2 ** 63 - 1) * 0.000000001) ..
                +((2 ** 63 - 1) * 0.000000001);
```

Ниже следуют примеры описаний вещественных типов с фиксированной точкой.

```
type Volt is delta 0.125 range 0.0 .. 255.0;


type Fraction is delta System.Fine_Delta range -1.0..1.0;        -- Ada95
```

```
                -- Fraction'Last = 1.0 - System.Fine_Delta
```

Последний пример показывает полезную способность вещественных типов с фиксированной точкой - четкое определение насколько тип должен быть точным. Например, это позволяет контролировать ошибки, возникающие при округлении.

2.3.3 Вещественные типы с десятичной фиксированной точкой

Следует учитывать, что поскольку от реализации компилятора Ады не требуется обязательное обеспечение поддержки вещественных величин с десятичной фиксированной точкой, то это может вызвать трудности при переносе программного обеспечения на другую систему или при использовании разных компиляторов.

Примером описания вещественного типа с десятичной фиксированной точкой может служить следующее:

```
type Money is delta 0.01 digits 15; -- десятичная фиксированная точка,
                                    -- здесь величина, задаваемая в delta,
                                    -- должна быть степенью 10
subtype Salary is Money digits 10;
```

2.3.4 Типы `Universal_Float` и `Root_Real`

Подобно тому как все целочисленные литералы принадлежат универсальному классу `Universal_Integer`, все вещественные численные литералы принадлежат универсальному классу `Universal_Float`. Этот универсальный тип совместим с любым вещественным типом с плавающей точкой и любым вещественным типом с фиксированной точкой. Такой подход избавляет от необходимости выполнения различных преобразований типов при описании вещественных констант и переменных.

Модель вещественной арифметики Ады основывается на анонимном типе `Root_Real`. Этот анонимный тип используется как базовый тип для всех вещественных типов. Тип `Root_Real` имеет точность, которая определяется значением константы `Max_Base_Digits` пакета *System* (`System.Max_Base_Digits`). Такой подход использован для облегчения переносимости программ.

2.3.5 Пакеты для численной обработки

Полное обсуждение поддержки численной обработки в Аде - весьма обширная тема. Поэтому здесь, чтобы указать "куда бежать дальше", мы только приведем список пакетов для численной обработки, которые предоставляются поставкой компиляора *GNAT*:

```
Ada.Numerics
Ada.Numerics.Aux
Ada.Numerics.Float_Random
Ada.Numerics.Discrete_Random


Ada.Numerics.Complex_Types
Ada.Numerics.Complex_Elementary_Functions
Ada.Numerics.Elementary_Functions


Ada.Numerics.Generic_Complex_Types
Ada.Numerics.Generic_Complex_Elementary_Functions
Ada.Numerics.Generic_Elementary_Functions


Ada.Numerics.Long_Complex_Types
```

```
Ada.Numerics.Long_Complex_Elementary_Functions
Ada.Numerics.Long_Elementary_Functions


Ada.Numerics.Long_Long_Complex_Types
Ada.Numerics.Long_Long_Complex_Elementary_Functions
Ada.Numerics.Long_Long_Elementary_Functions


Ada.Numerics.Short_Complex_Types
Ada.Numerics.Short_Complex_Elementary_Functions
Ada.Numerics.Short_Elementary_Functions
```

Следует заметить, что пакет *Ada.Numerics.Aux*, который указан выше, не предназначен для непосредственного использования в программах пользователя, и упоминается только с целью полноты показанного выше списка.

2.4 Преобразование численных типов

Поскольку тип `Float` и тип `Integer` - различные типы, то Ада не допускает смешивания величин этих типов в одном выражении. Однако, встречаются случаи, когда нам необходимо комбинировать значения этих типов. В таких ситуациях нам необходимо производить преобразование значений одного численного типа в значения другого численного типа.

Ада позволяет явно указывать необходимость преобразования значения типа `Float` в значение типа `Integer`, и наоборот. Для выполнения такого преобразования используется синтаксис подобный синтаксису вызова функции:

```
X : Integer := 4;
Y : Float;

Y := Float(X);


 ...

X := Integer(Y);
```

В этом случае компилятор, во время трансляции, добавит необходимый код для преобразования типов. Такое преобразование типов всегда возможно, и будет успешным если значение результата не будет нарушать границ допустимого диапазона значений.

Следует заметить, что при преобразовании вещественного значения `Float` в целое значение `Integer` Ада использует традиционные для математики правила округления, то есть:

Значение `Float`	Округленное значение `Integer`
1.5	2
1.3	1
-1.5	-2
-1.3	-1

2.5 Перечислимые типы

До настоящего момента были рассмотрены типы данных которые представляли численные значения (целые и вещественные числа). Однако, при решении многих практических задач, важное значение имеет понятие перечислимого типа. Перечислимыми типами называют такие типы данных значения которых перечислены, то есть представлены списком некоторых значений. Перечислимый тип полезен когда необходимо представить фиксированное множество значений, которые не являются числами. Примером может служить представление дней недели или месяцев в году.

2.5.1 Описание перечислимого типа

Перечислимый тип описывается путем предоставления списка всех возможных значений данного типа в виде идентификаторов (другими словами, перечислением всех возможных значений данного типа). Например:

```
type Computer_Language is (Assembler, Cobol, Lisp, Pascal, Ada);
type C_Letter_Languages is (Cobol, C);
```

После такого описания типов, константы и переменные этих типов могут быть описаны следующим образом:

```
A_Language      : Computer_Language;
Early_Language  : Computer_Language := Cobol;
First_Language  : constant Computer_Language := Assembler;
Example         : C_Letter_Language := Cobol;
```

Необходимо заметить, что порядок перечисления значений типа, при описании перечислимого типа, имеет самостоятельное значение - он устанавливает отношение порядка следования значений перечислимого типа, которое используется атрибутами ***'First***, ***'Last***, ***'Pos***, ***'Val***, ***'Pred***, ***'Succ*** (см "Атрибуты типов"), а также при сравнении величин перечислимого типа. Так, для типа `Computer_Language`, описанного в примере выше, значение `Assembler` будет меньше чем значение `Cobol`.

В одной программе допускается использование одного и того же перечислимого литерала в описаниях различных перечислимых типов. Так, в показанном выше примере, литерал `Cobol` встречается при описании двух разных типов: `Computer_Language` и `C_Letter_Languages`. Такие литералы называют **совмещенными** (или перегруженными). При этом, Ада, в большинстве случаев, распознает одинаковые перечислимые литералы различных перечислимых типов. В случаях, когда это не возможно - необходимо использовать квалификацию типов.

Рассмотрим следующий пример:

```
type Primary is (Red, Green, Blue);
type Rainbow is (Red, Yellow, Green, Blue, Violet);
...
for I in Red..Blue loop ...      -- это двусмысленно
```

Здесь, компилятор не может самостоятельно определить к какому из двух типов (`Primary` или `Rainbow`) принадлежит диапазон значений `Red..Blue` переменной цикла `I` (литералы `Red` и `Blue` - совмещены). Поэтому, в подобных случаях, нам необходимо точно указывать требуемый тип, используя квалификацию типа:

```
for I in Rainbow'(Red)..Rainbow'(Blue) loop ...
for I in Rainbow'(Red)..Blue loop ...     -- необходима только одна квалификация
for I in Primary'(Red)..Blue loop ...
```

Квалификация типа не изменяет значения типа и не выполняет никаких преобразований типа. Она только информирует компиляор о том какой тип, в данном случае, подразумевает программист.

При описании перичислимых типов, для указания значений перечислимого типа также как и символические имена, допускается использовать символьные константы. В этом случае перечислимый тип будет символьным (см. "Символьные типы Ады").

В заключение обсуждения описания перечислимых типов, заметим, что перечислимые и целочисленные типы называют **дискретными типами**, потому что они описывают множества

упорядоченных дискретных значений. Вещественные числа не могут считаться дискретными поскольку между двумя любыми вещественными числами располагается бесконечное множество вещественных чисел (естественно, теоретически).

2.5.2 Предопределенный логический тип `Boolean`

В Аде, предопределенный логический тип `Boolean` описывается как перечислимый тип в пакете *Standard*:

```
type  Boolean  is  (False, True);
```

Таким образом, переменные логического типа `Boolean` могут принимать только одно из двух значений: `True` (истина) или `False` (ложь).

Примечательно, что предопределеннный логический тип `Boolean` имеет специальное предназначение. Значения этого типа используются в условных инструкциях языка Ада ("**`if ...`** ", "**`exit when ...`** ", ...). Это подразумевает, что если вы пытаетесь описать свой собственный логический тип `Boolean`, и описываете его точно также как и предопределенный тип `Boolean` (полное имя предопределенного логического типа - `Standard.Boolean`), то вы получаете абсолютно самостоятельный тип(!). В результате, вы не можете использовать значения, описанного вами типа `Boolean`, в условных инструкциях языка Ада, которые ожидают только тип `Standard.Boolean`.

Значения предопределенного логического типа `Standard.Boolean` возвращают знаки операций сравнения:

```
=     -- равно
/=    -- не равно
<     -- меньше
<=    -- меньше или равно
>     -- больше
>=    -- больше или равно
```

Для обработки значений типа `Boolean` могут быть использованы следующие знаки операций:

```
and         -- логическое И:    вычисляются и левая, и правая часть выражения
and then    -- логическое И:    правая часть выражения вычисляется, если

         --         результат вычисления левой части - True


or          -- логическое ИЛИ:  вычисляются и левая, и правая часть выражения
or else     -- логическое ИЛИ:  правая часть выражения вычисляется, если

         --         результат вычисления левой части - False


xor         -- исключающее ИЛИ
not         -- отрицание (инверсия); унарная операция
```

Обычно, при вычислении значений логических выражений, компилятор сам определяет последовательность вычисления каждого логического значения. При этом, производится вычисление всех логических переменных, указанных в логическом выражении. Тем кто знаком с языком Паскаль следует заметить, что такой подход отличается от правил принятых в современных диалектах Паскаля, где, для повышения производительности, определение значения результата всего логического выражения может быть выполнено сокращенно, в зависимости от предварительных результатов обработки выражения. Например, при определении значения результата следующего логического выражения

```
(B /= 0) and (A/B > 0)
```

в случае, когда значение B равно нулю будет возникать ошибка деления на ноль. Причина в том, что значение части выражения, расположенной справа от "**and**", вычисляется всегда, не зависимо от значения результата полученного при вычислении (B /= 0).

Чтобы избежать подобных ошибок, а также в целях увеличения производительности, необходимо производить вычисление значений логических переменных в определенной последовательности и прекращать ее как только результат всего выражения уже определен. Для этого можно использовать "**and then**" вместо "**and**", и "**or else**" вместо "**else**", указывая порядок обработки логического выражения явно:

```
(B /= 0) and then (A/B > 0)
```

В этом случае, обработка выражения справа от "**and then**" будет производиться только в случае когда B не равно нулю, т.е. результат слева от "**and then**" - True.

Можно переписать предыдущий пример с использованием "**or else**". Тогда, обработка логического выражения будет завершена в случае если значение слева от "**or else**" вычислено как True:

```
(B = 0) or else (A/B <= 0)
```

В заключение обсуждения логического типа отметим, что Ада не позволяет одновременное использование "**and**" ("**and**" или "**and then**"), "**or**" ("**or**" или "**or else**") и "**xor**" в одном выражении не разделенном скобками. Это уменьшает вероятность разночтения содержимого сложного логического выражения. Например:

```
(A < B) and (B > C) or (D < E)    -- запрещено
((A < B) and (B > C)) or (D < E)  -- разрешено
```

2.5.3 Символьные типы Ады (Character, Wide_Character)

Ада имеет два предопределенных перечислимый типа, Character и Wide_Character, для поддержки обработки символьных значений. Согласно стандарта, любой перичислимый тип который содержит хотя бы один символьный литерал является символьным типом.

Оригинальный стандарт Ada83 описывал 7-битный тип Character. Еще до появления стандарта Ada95, это ограничение было ослаблено, но оставалось принудительным для старых компиляторов (например таких как компилятор *Meridian Ada*). Это создавало трудности при попытках отобразить графические символы на PC, поскольку для отображения символов с кодами большими чем ASCII-127 приходилось использовать целые числа. Такая поддержка обеспечивалась за счет специальных подпрограмм предоставляемых разработчиками соответствующего компилятора.

В настоящее время, предопределенный символьный тип Character предусматривает 256 различных символьных значений (то есть, является 8-битным), и основывается на стандарте ISO-8859-1 (Latin-1).

Некоторые символы не имеют непосредственно печатаемого значения (первые 32 символа). Такие символы используются в качестве управляющих (примером может служить символ CR - возврат каретки). Для обращения к таким символам можно использовать пакет *ASCII*, который является дочерним пакетом пакета *Standard* (благодаря этому, нет необходимости указывать

пакет *ASCII* в спецификаторах контекста **`with`** и/или **`use`**). Например, для обращения к символу возврат каретки можно использовать: `ASCII.CR`. Однако, пакет *ASCII* содержит только первые 128 символов и считается устаревшим, и возможно, что в будущем он будет удален. Поэтому, вместо старого пакета *ASCII* рекомендуется использовать пакет *Ada.Characters.Latin_1*, который предоставляет 256 символов. Следовательно, используя пакет *Ada.Characters.Latin_1*, к символу возврата каретки можно обратиться следующим образом: `Ada.Characters.Latin_1.CR`.

Предопределенный символьный тип `Wide_Character` основывается на стандарте ISO-10646 Basic Multilingual Plane (BMP) и предусматривает 65336 различных символьных значений (использует 16 бит).

Также, Ада предоставляет пакет *Ada.Characters.Handling*, предлагающий набор полезных подпрограмм символьной обработки.

Система компилятора *GNAT* предоставляет дополнительный пакет *Ada.Characters.Wide_Latin_1*, который описывает символьные значения типа `Wide_Character` соответствующие кодировке Latin_1.

Таким образом, для работы с символьными значениями, в Аде представлены следующие пакеты:

```
Standard.ASCII                -- предоставляет только первые 128 символов
                              -- (считается устаревшим)

Ada.Characters                --
Ada.Characters.Latin_1        -- предоставляет 256 символов ISO-8859-1 (Latin-1)
Ada.Characters.Handling       -- предоставляет подпрограммы символьной обработки

Ada.Characters.Wide_Latin_1   -- дополнительный пакет из поставки
                                 -- системы компилятора GNAT
```

Следует заметить, что пакет *ASCII* считается устаревшим и его не рекомендуется использовать при написании новых программ. Вместо него необходимо использовать стандартный пакет *Ada.Characters.Latin_1*.

Также следует заметить, что стандарт не требует полноценного обеспечения поддержки национальных кодировок, которые отличны от кодировки Latin-1. Возможность предоставления такой поддержки возлагается на разработчиков соответствующего компилятора. Другими словами, в настоящее время, обеспечение поддержки кириллических кодировок не регламентируется стандартом.

2.6 Типы и подтипы

Как уже говорилось, концепция типа является в Аде основополагающим фактором создания надежного программного обеспечения. Предположим, что у нас есть два целочисленных типа, которые как-то характеризуют "звоночки" и "свисточки", и мы никак не хотим смешивать эти понятия. Нам необходимо, чтобы компилятор имел возможность предупредить нас: "Ба, да вы пытаетесь смешать выши "звоночки" и "свисточки"! Что Вы в действительности подразумеваете?". Это выглядит излишними осложнениями для людей, которые используют другие языки программирования со слабой типизацией данных. Размышления над тем какие данные необходимо представить, описание различных типов данных и правил конвертирования типов требуют некоторых усилий. Однако, такие усилия оправдываются тем, что как только это сделано, компилятор сможет помочь отыскать разные глупые ошибки.

Бывают случаи, когда нам необходимо указать, что какие-то переменные могут хранить только

какое-то ограниченное число значений предоставляемых определенным типом данных. При этом, мы не хотим описывать самостоятельный новый тип данных, поскольку это потребует явного указания правила преобразования типа при взаимодействии со значениями, которые имеют оригинальный тип данных, то есть нам необходима возможность смешивать различные типы, которые, по своей сути, являются одинаковыми сущностями, но при этом имеют некоторые различия в своих характеристиках. К тому же, мы хотим сохранить преимущества проверки корректности наших действий, которые предоставляет компилятор. В таких случаях, Ада позволяет нам описывать подтипы (**subtype**) существующих типов данных (это соответствует типам определяемым пользователем в Паскале).

Общий синтаксис объявления подтипа имеет вид:

```
subtype Name_1 is Base_Type;  -- в данном случае, Name_1 является
                              -- синонимом типа Base_Type

subtype Name_2 is Base_Type range Lowerbound..Upperbound;
```

Примеры объявления подтипов приводятся ниже:

```
type Processors is (M68000, i8086, i80386, M68030, Pentium, PowerPC);
subtype Old_Processors is Processors range M68000..i8086;
subtype New_Processors is Processors range Pentium..PowerPC;


subtype Data is Integer;
subtype Age is Data range 0..140;
subtype Temperatures is Float range -50.0..200.0;
subtype Upper_Chars is Character range 'A'..'Z';
```

Подтип, по своей сути, является той же самой сущностью, что и оригинальный тип, но при этом он может иметь ограниченный диапазон значений оригинального типа. Значения подтипа могут использоваться везде, где могут использоваться значения оригинального типа, а также значения других подтипов этого оригинального типа При этом, любая попытка присваивания переменной такого подтипа значения выходящего за границы указанного для этого подтипа диапазона допустимых значений будет приводить к исключительной ситуации *Constraint_Error* (проще говоря - ошибке программы). Такой подход облегчает обнаружение ошибок, а также, позволяет отделить обработку ошибок от основного алгоритма программы.

```
My_Age  : Age;
Height  : Integer;


Height := My_Age;   -- глупо, но никогда не вызывает проблем


My_Age := Height;   -- может вызвать проблемы, когда значение типа Height
                    -- будет за пределами диапазона значений My_Age (0..140),
                    -- но при этом остается совметимым
```

Чтобы избежать генерацию исключительной ситуации, можно использовать проверки принадлежности диапазону ("**in**" и/или "**not in**"). Например:

```
I : Integer;
N : Natural;


   ...


if  I in Natural  then
   N := I
else
   Put_Line ("I can't be assigned to N!");
```

```
        end if;

              . . .
```

Реально, все типы Ады являются подтипами анонимных типов, рассматриваемых как их **базовые типы**. Поскольку базовые типы анонимны, то на них нельзя ссылаться по имени. При этом, для получения базового типа можно использовать атрибут **`'Base`**. Например, `Integer'Base` - это базовый тип для `Integer`. Базовые типы могут иметь или могут не иметь диапазон значений больший чем их подтипы. Это имеет значение только в выражениях вида "`A * B / C`" которые, при вычислении промежуточных значений, используют базовый тип. То есть, результат "`A * B`" может выходить за пределы значений типа не приводя к генерации исключительной ситуации если общий результат вычисленного значения всего выражения будет находиться в допустимом диапазоне значений для данного типа.

Таким образом, необходимо заметить, что проверка допустимости диапазона производится только для значений подтипов, а для значений базовых анонимных типов такая проверка не производится. При этом, чтобы результат вычислений был математически корректен, всегда производится проверка на переполнение.

2.7 Производные типы

Мы уже рассмотрели, что подтипы остаются совместимыми со своими базовыми типами. Однако, нам может понадобиться создать абсолютно новый тип, который не будет ассоциироваться ни с каким другим типом вообще, в том числе и с оригинальным типом. Такая концепция типа сильно отличается от того, что принято в других языках программирования, и, даже, в прародителе Ады - Паскале.

Для выполнения поставленной задачи, мы производим новый тип от другого типа, используя подобный синтаксис:

```
        type Child_Type is new Parent_Type;
```

Такое описание создает новый тип данных - `Child_Type`, при этом `Parent_Type` - это тип-предок для типа `Child_Type`, а тип `Child_Type` - это производный тип от типа `Parent_Type`.

В данном случае, тип `Child_Type` будет обладать такими же характеристиками что и его тип-предок `Parent_Type`: у него такой же диапазон допустимых значений как и у типа-предка, для него допустимы те же операции, которые допустимы для типа-предка (говорят, что тип `Child_Type` унаследовал операции от типа-предка `Parent_Type`).

Однако, в этом случае, производный тип `Child_Type` - это абсолютно самостоятельный и независимый тип данных. Он не совместим ни с каким другим типом, включая тип `Parent_Type`, от которого он был произведен, и другими типами, производными от типа `Parent_Type`. Это подразумевает, что при необходимости комбинирования значений типа `Child_Type` со значениями типа `Parent_Type` требуется выполнять преобразование типов. В Аде, разрешается выполнять преобразование значений производного типа в значения типа-предка и наоборот.

Для того, чтобы преобразовать значение одного типа в значение другого типа необходимо указать имя типа к которому требуется выполнить преобразование:

```
Parent  : Parent_Type;
Child   : Child_Type := Child_Type (Parent);  -- конвертирует значение Parent,
                                              -- имеющее тип Parent_Type
                                                                  -- в значение типа Child_Type
```

Описание производного типа может указывать ограничение диапазона значений типа-предка, например:

```
type Child_Type is new Parent_Type range Lowerbound..Upperbound;
```

В этом случае диапазон значений производного типа `Child_Type` будет ограничен значениями нижней границы диапазона (`Lowerbound`) и верхней границы диапазона (`Upperbound`).

Механизм производства новых типов данных из уже существующих типов позволяет создавать целые семейства родственных типов, которые обычно называют классами. Так, например, тип `Integer` принадлежит к целому классу целочисленных типов. Класс целочисленных типов является, в свою очередь, подмножеством более обширного класса дискретных типов.

Основной смысл использования производных типов заключается в том, что для определенного типа данных уже существует определенный набор примитивных операций и этот набор операций можно унаследовать от такого типа при производстве от него нового типа данных.

Можно создать новый тип данных и затем описать для него идентичный набор операций. Однако, при производстве нового типа данных от уже существующего типа, производный тип автоматически наследует версии примитивных операций, описанные для типа-предка. Таким образом, нет необходимости переписывать код заново.

Например, класс дискретных типов предусматривает атрибут `'First`, который наследуется всеми дискретными типами. Класс целочисленных типов добавляет к унаследованным от класса дискретных типов операциям знак операции арифметического сложения "+". Эти механизмы более полно рассматриваются при обсуждении тэговых типов.

Производные типы используются в случаях когда моделирование определенного объекта предполагает, что тип-предок не соответствует нашим потребностям, или тогда, когда мы желаем разделить объекты в различные, несмешиваемые классы.

```
type Employee_No is new Integer;
type Account_No  is new Integer range 0..999_999;
```

Здесь `Employee_No` и `Account_No` различные и не смешиваемые типы, которые нельзя комбинировать между собой без явного использования преобразования типов. Производные типы наследуют все операции объявленные для базового типа. Например, если была объявлена запись которая имела процедуры `Push` и `Pop`, то производный тип автоматически унаследует эти процедуры.

Другое важное использование производных типов - это создание переносимого кода. Ада разрешает нам создавать новые уровни абстракции, на один уровень выше чем, скажем, абстракция целого числа на последовательности битов.

Это определяется использованием производных типов, без типа-предка.

```
type Name is range <некоторый_диапазон_значений>;
```

Например,

```
type Data is range 0..2_000_000;
```

В этом случае ответственность, за выбор подходящего размера для целого, ложится на компилятор. Таким образом, для PC, этот размер может быть 32 бита, что эквивалентно типу `Long_Integer`, а для рабочей станции Unix, этот размер можт остаться равным 32-битному целому, но это будет эквивалентно типу `Integer`. Такое предоставление компилятору возможности выбирать освобождает программиста от обязанности выбирать. Поэтому перекомпиляция программы на другой машине не требует изменения исходного текста.

2.8 Атрибуты

Ада предусматривает специальный класс предопределенных операций, которые позволяют в удобной форме определять и использовать различные характеристики типов и экземпляров объектов. Они называются атрибутами.

Указание имени требуемого атрибута производится в конце имени экземпляра объекта (переменной, константы) или имени типа с использованием символа одинарных кавычек.

```
имя_типа'имя_атрибута
имя_экземпляра_объекта'имя_атрибута
```

Некоторыми из атрибутов для дискретных типов являются:

```
type Processors is (M68000, i8086, i80386, M68030, Pentium, PowerPC);


Integer'First             -- наименьшее целое Integer
Integer'Last              -- наибольшее целое Integer
Processors'Succ(M68000)   -- последующее за M68000 в типе
Upper_Chars'Pred('C')     -- предшествующее перед 'C' в типе ('B')
Integer'Image(67)         -- строка " 67"

                    -- пробел для '-'
Integer'Value("67")       -- целое значение 67
Processors'Pos(M68030)    -- позиция M68030 в типе
                                    -- (3, первая позиция - 0)
```

Простым примером использования атрибутов, - для улучшения переносимости программы, - может служить следующий пример описания подтипа `Positive`, предоставляющего положительные целые числа:

```
subtype Positive is Integer range 1..Integer'Last;
```

Здесь мы получаем размер максимального положительного целого не зависимо от любых системно зависимых свойств.

В Ada83 для не дискретных типов, - таких как `Float`, `Fixed`, всех их подтипов и производных от них типов, - концепции **`'Pred`** и **`'Succ`** - не имеют смысла. В Ada95 - они присутствуют. Все другие атрибуты скалярных типов также справедливы и для вещественных типов.

3. Управляющие структуры

Управляющие структуры любого языка программирования предназначены для описания алгоритмов программ, или, другими словами, для описания последовательности тех действий которые выполняет программа во время своей работы.

Управляющие структуры языка Ада по своему стилю и назначению подобны тем конструкциям, которые встречаются в большинстве других современных языках программирования. Однако,

при этом присутствуют и некоторые отличия.

Как и язык в целом, управляющие структуры языка Ада, были разработаны так, чтобы обеспечить максимальную читабельность. Все управляющие структуры языка Ада можно разделить на простые и составные инструкции. Все инструкции языка Ада должны завершаться символом точки с запятой. Все составные инструкции завершаются конструкцией вида "**end** *что-нибудь* ;".

3.1 Пустая инструкция

В отличие от многих других языков программирования, в Аде явно присутствует инструкция, которая не вызывает каких-либо действий. Такая инструкция называется пустой, и она имеет следующий вид:

```
null;
```

Пустая инструкция используется в тех случаях когда не требуется выполнение каких-либо действий, но согласно синтаксиса языка должна быть записана хотя бы одна инструкция.

3.2 Инструкция присваивания

Инструкция присваивания используется в Аде для установки и изменения значений переменных. Она использует операцию присваивания, и в общем случае имеет следующий вид:

```
result := expression;
```

Операция присваивания "**:**=" разделяет инструкцию присваивания на левую и правую части (между символами двоеточия и знак равенства, пробелы - не допустимы!). В левой части записывается имя переменной (*result*) содержимому которой будет производится присваивание нового значения. Следует заметить, что в левой части может располагаться имя только одной переменной. В правой части записывается выражение (*expression*) результат вычисления которого становится новым значением переменной *result*. Выражение *expression*, расположенное в правой части, может быть одиночной переменной или константой, может содержать переменные, константы, знаки операций и вызовы функций. Тип переменной определяемой как *result* должен быть совместим по присваиванию с типом результата вычисления выражения *expression*.

Выполнение присваивания разделяется на несколько действий. Сначала производится вычисление имени переменной *result* и результата выражения *expression* (порядок следования этих действий не регламентирован стандартом языка). После этого, в случае успеха, для переменных скалярных типов проверяется принадлежность значения результата вычисления выражения *expression* подтипу переменной. Если проверка успешна, то значение результата вычисления выражения *expression* становится новым значением содержимого переменной *result*. При этом старое значение содержимого *result* - теряется. Иначе, в случае какой-либо неудачи, возбуждается исключение ошибки ограничения или, проще говоря, - ошибка, а значение переменной *result* остается без изменений.

Приведем несколько примеров инструкций присваивания:

```
A := B + C
X := Y
```

Следует заметить, что в Аде, в результате выполнения присваивания производится изменение

только содержимого *result* (значение содержимого переменной указанной в левой части). Необходимо также подчеркнуть, что операция присваивания в Аде, в отличие от языков C/C++, не возвращает значение и не обладает побочными эффектами. Кроме того, напомним, что операция присваивания, в Аде, не допускает совмещение, переименование и использование псевдонимов, а также она запрещена для лимитированных типов.

3.3 Блоки

Блок содержит последовательность инструкций, перед которой может располагаться раздел описаний (все описания локальны для блока и не доступны вне блока). За последовательностью инструкций могут следовать обработчики исключений (обработка исключений в Аде рассматривается позже). В общем случае инструкция блока Ады имеет вид:

```
declare

    -- локальные описания

begin

    -- последовательность инструкций

exeption

    -- обработчики исключений

end;
```

Блок может иметь имя. Для этого перед инструкцией блока записывается идентификатор, за которым ставится двоеточие. При именовании блока, имя блока должно указываться после **end** завершающего блок:

```
Some_Block:
declare
    X : Integer;
begin
    X := 222 * 333;
    Put(X);
end Some_Block;
```

3.4 Условные инструкции `if`

Для организации условного выполнения последовательностей алгоритмических действий (то есть, построения разветвляющихся алгоритмов), в Аде могут использоваться условные инструкции **`if`**.

Каждая инструкция **`if`** заканчивается конструкцией "**`end if`**".

```
if <логическое_выражение> then

    -- последовательность инструкций

end if;

if <логическое_выражение> then
```

```
    -- последовательность инструкций 1
else
    -- другая последовательность инструкций 2
end if;
```

В первом примере, приведенном выше, последовательность инструкций, описывающая алгоритмические действия, будет выполнена только в случае когда результат вычисления логического выражения будет иметь значение `True`. Во втором примере, в случае когда результат вычисления логического выражения - `True` будет выполняться "`последовательность инструкций 1`", в противном случае - "`последовательность инструкций 2`".

Для сокращения инструкций вида "**`else if ...`** ", и в целях улучшения читабельности, введена конструкция **`elsif`**, которая может быть использована столько раз, сколько это будет необходимо.

```
if <логическое_выражение> then
    -- последовательность инструкций 1
elsif <логическое_выражение> then
    -- последовательность инструкций 2
elsif <логическое_выражение> then
    -- последовательность инструкций 3
else
    -- последовательность инструкций
end if;
```

В этой форме инструкции **`if`**, заключительная конструкция **`else`** - опциональна.

Необходимо также заметить, что результат вычисления логического выражения всегда должен иметь предопределенный тип `Standard.Boolean`.

3.5 Инструкция выбора `case`

Еще одним средством позволяющим строить разветвляющиеся алгоритмы является инструкция выбора **`case`**.

Инструкция выбора **`case`** должна предусматривать определенное действие для каждого возможного значения переменной селектора (переключателя). В случаях, когда невозможно перечислить все значения переменной селектора, нужно использовать метку **`others`**.

Каждое значение выбора может быть представлено как одиночное значение (например, 5), как

диапазон значений (например, 1..20), или как комбинация, состоящая из одиночных значений и/или диапазонов значений, разделенных символом ' | '.

Каждое значение выбора должно быть статическим значением, то есть оно должно быть определено компилятором во время компиляции программы.

```
case  выражение  is
    when значение_выбора => действия
    when значение_выбора => действия
      . . .
    when others => действия
end case;
```

Важные примечания:

- "*выражение*", в инструкции **case**, должно быть дискретного типа
- метка **others** обязательна в инструкции **case** тогда, когда инструкции **when** не перечисляют всех возможных значений селектора.

```
case Letter is
    when 'a'..'z'| 'A'..'Z' => Put ("letter");
    when '0'..'9'          => Put ("digit! Value is"); Put (letter);
    when ''' | '"' | '`'   => Put ("quote mark");
    when '&'               => Put ("ampersand");
    when others            => Put ("something else");
end case;
```

В некоторых случаях, в качестве действий, указываемых для метки **others**, может использоваться пустая инструкция **null**:

```
    . . .
    when others  => null;  -- ничего не делать
        . . .
```

3.6 Организация циклических вычислений

При решении реальных задач часто возникает необходимость в организации циклических вычислений. Все конструкции организации циклических вычислений в Аде имеют форму "**loop** ... **end loop**" с некоторыми вариациями. Для выхода из цикла может быть использована инструкция **exit**.

3.6.1 Простые циклы (loop)

Примером простейшего цикла может служить бесконечный цикл. Обычно он используется совместно с инструкцией **exit**, рассматриваемой позже.

```
loop

    -- инструкции тела цикла

end loop;
```

3.6.2 Цикл `while`

Во многих случаях, прежде чем выполнять действия которые описываются инструкциями тела цикла, необходимо проверить какое-либо условие. Для таких случаев Ада предусматривает конструкцию цикла **`while`**.

Цикл **`while`** идентичен циклу **`while`** в языке Паскаль. Проверка условия выполнения цикла производится до входа в блок инструкций составляющих тело цикла. При этом, если результат вычисления логического выражения будет `True`, то будет выполнен блок инструкций тела цикла. В противном случае, тело цикла - не выполняется.

```
while логическое_выражение loop

   -- инструкции тела цикла

end loop;
```

Необходимо заметить, что результат вычисления логического выражения должен иметь предопределенный тип `Standard.Boolean`

3.6.3 Цикл `for`

Еще одним распространенным случаем является ситуация когда необходимо выполнить некоторые действия заданное количество раз, то есть организовать счетный цикл. Для этого Ада предусматривает конструкцию цикла **`for`**.

Конструкция цикла **`for`** Ады аналогична конструкции цикла **`for`**, представленной в языке Паскаль.

Существует несколько правил использования цикла **`for`**:

- тип переменной-счетчика цикла **`for`** определяется типом указываемого диапазона значений счетчика, и должен быть дискретного типа, вещественные значения - недопустимы
- счетчик не может быть модифицирован в теле цикла, другими словами - счетчик доступен только по чтению
- область действия переменной-счетчика распространяется только на тело цикла

Примечательно также, что тело цикла не будет выполняться если при указании диапазона значений переменной-счетчика величина значения "нижней границы" будет больше чем величина значения "верхней границы".

```
for счетчик in диапазон_значений_счетчика loop

   -- инструкции тела цикла

end loop;



for  Count in 1..20 loop
   Put (Count);
end loop;
```

Возможен перебор значений диапазона в обратном порядке:

```
for счетчик in reverse диапазон_значений_счетчика loop
```

```
    -- инструкции тела цикла

end loop;



for Count in reverse 1..20 loop
    Put (Count);
end loop;
```

Любой дискретный тип может использоваться для указания диапазона значений переменной-счетчика.

```
declare
    subtype List is Integer range 1..10;

begin
    for Count in List loop
        Put (Count);
    end loop;
end;
```

Здесь, тип `List` был использован для указания диапазона значений переменной-счетчика `Count`. Подобным образом также можно использовать любой перечислимый тип.

3.6.4 Инструкции `exit` и `exit when`

Инструкции **`exit`** и **`exit when`** могут быть использованы для преждевременного выхода из цикла. При этом, выполнение программы будет продолжено в точке непосредственно следующей за циклом. Два варианта, показанных ниже, имеют одинаковый эффект:

```
loop

    -- инструкции тела цикла

    if логическое_выражение then
        exit;
    end if;
end loop;



loop

    -- инструкции тела цикла

    exit when логическое_выражение;
end loop;
```

3.6.5 Именованые циклы

Инструкции преждевременного выхода из цикла **`exit`** и **`exit when`**, обычно, осуществляют выход из того цикла, который непосредственно содержит данную инструкцию. Однако, мы можем именовать циклы и модифицировать инструкцию выхода из цикла так, чтобы

осуществлять выход сразу из всех вложенных циклов. Во всех случаях, следующая выполняемая инструкция будет следовать сразу за циклом из которого был осуществлен выход.

```
outer_loop:
loop


   -- инструкции


   loop


      -- инструкции


      exit outer_loop when логическое_выражение;
   end loop;
end loop outer_loop;
```

Примечательно, что в случае именованого цикла **`end loop`** также необходимо именовать меткой.

3.7 Инструкция перехода `goto`

Инструкция перехода **`goto`** предусмотрена для использования в языке Ада, в исключительных ситуациях, и имеет следующий вид:

```
goto Label;

<<Label>>
```

Использование инструкции **`goto`** очень ограничено и четко осмысленно. Вы не можете выполнить переход внутрь условной инструкции **`if`**, внутрь цикла (**`loop`**), или, как в языке Паскаль, за пределы подпрограммы.

Вообще, при таком богатстве алгоритмических средств Ады, использование **`goto`** едва-ли можно считать оправданным.

4. Массивы (*array*)

Понятие массива подразумевает один из механизмов структурирования данных и является одним из способов построения составных типов данных. В сущности, массив - это набор данных идентичного типа. Как правило, массиву дается какое-то имя, которое будет обозначать весь набор данных, и механизм индексации, позволяющий обращаться к индивидуальным элементам набора. На сегодняшний день, большинство языков программирования предусматривают возможность работы с различного типа массивами.

Ада предоставляет массивы подобно языку Паскаль, и, при этом, добавляет некоторые новые полезные особенности. Неограниченные и динамические массивы, атрибуты массивов - вот некоторые из предлагаемых расширений.

4.1 Простые массивы

4.1.1 Описание простого массива

В общем случае, при объявлении массива, сначала производится описание соответствующего типа. Затем, экземпляр массива может быть создан используя описание этого типа.

```
type Stack is array (1..50) of Integer;
Calculator_Workspace : Stack;


type Stock_Level is Integer range 0..20_000;
type Pet is (Dog, Budgie, Rabbit);
type Pet_Stock is array(Pet) of Stock_Level;


Store_1_Stock : Pet_Stock;
Store_2_Stock : Pet_Stock;
```

В приведенном выше примере, тип `Stack` - это массив из 50-ти целочисленных элементов типа `Integer`, а `Calculator_Workspace` - это переменная типа `Stack`. Еще одним описанием массива является тип `Pet_Stock`. При этом, тип `Pet_Stock` - это массив элементов типа `Stock_Level`, а для индексирования элементов массива `Stock_Level` используется перечислимый тип `Pet`. Переменные `Store_1_Stock` и `Store_2_Stock` - это переменные типа `Pet_Stock`.

Общая форма описания массива имеет следующий вид:

```
type <имя_массива> is array (<спецификация_индекса>) of <тип_элементов_массива>;
```

Необходимо заметить:

- спецификация индекса может быть типом (например, `Pet`)
- спецификация индекса может быть диапазоном (например, `1..50`)
- значения индекса должны быть дискретного типа

4.1.2 Анонимные массивы

Массив может быть объявлен непосредственно, без использования предопределенного типа:

```
No_Of_Desks : array(1..No_Of_Divisions) of Integer;
```

В этом случае массив будет называться анонимным (поскольку он не имеет явного типа) и он будет несовместим с другими массивами - даже такими, которые описаны таким же самым образом. Кроме того, такие массивы не могут быть использованы как параметры подпрограмм. В общем случае рекомендуется избегать использования анонимных массивов.

4.1.3 Организация доступа к отдельным элементам массива

Организацию доступа к отдельным элементам массива проще всего продемонстрировать на простых примерах. Так для обращения к значению элемента массива `Store_1_Stock`, описанного ранее, можно использовать:

```
if Store_1_Stock(Dog) > 10 then ...
```

В приведенном примере производится чтение значения элемента массива `Store_1_Stock` (доступ по чтению).

Для сохранения значения в элементе массива `Store_2_Stock` (доступ по записи) можно использовать:

```
Store_2_Stock(Rabbit) := 200;
```

Необходимо отметить, что в обоих случаях доступ к элементу массива в Аде внешне никак не отличается от вызова функции.

4.1.4 Агрегаты для массивов

В общем случае, агрегат массива - это совокупность значений для каждого элемента массива. Использование агрегатов позволяет выполнять одновременное присваивание значений всем элементам массива в эффективной и элегантной форме.

Рассмотрим следующий пример:

```
Store_1_Stock := (5, 4, 300);
```

В данном случае, присваивание значений элементам массива `Store_1_Stock` выполняется с помощью агрегата. Следует учесть, что в этом примере значения в агрегате присваиваются в порядке соответствующем следованию элементов в массиве. Такая нотация называется **позиционной** или **неименованой**, а такой агрегат - **позиционный** или **неименованый агрегат**.

Кроме позиционной нотации, возможно использование **именованой** нотации. В этом случае именуется каждый индивидуальный элемент массива. Используя именованую нотацию, предыдущий пример можно переписать следующим образом:

```
Store_1_Stock := (Dog => 5, Budgie => 4, Rabbit => 300);
```

Такой вид агрегата называют **именованым агрегатом**.

Приведем еще один пример именованого агрегата:

```
Store_1_Stock := (Dog | Budgie => 0, Rabbit => 100);
```

В пределах одного агрегата, Ада допускает использовать только один вид нотации. Это означает, что комбинирование позиционной и именованой нотации в одном агрегате - не допустимо и будет вызывать ошибку компиляции. Например:

```
Store_1_Stock := (5, 4, Rabbit => 300);    -- это недопустимо!
```

В агрегате может указываться диапазон дискретных значений:

```
Store_1_Stock := (Dog..Rabbit => 0);
```

Агрегаты обоих видов удобно использовать в описаниях:

```
Store_1_Stock:  Pet_Stock := (5, 4, 300);
```

С агрегатами массивов разрешается использование опции **`others`**, которая практически полезна при установке всех элементов массива в какое-либо предопределенное значение. Стоит учесть, что в таких случаях часто требуется квалификация типа.

```
New_Shop_Stock  : Pet_Stock := (others := 0);
```

Рассмотрим следующие описания:

```
declare
    type Numbers1 is array(1..10) of Integer;
    type Numbers2 is array(1..20) of Integer;
    A : Numbers1;
    B : Numbers2;
begin
    A := (1, 2, 3, 4, others => 5);
end;
```

Заметьте, что в данном случае опция **`others`** используется вместе с позиционной нотацией. Поэтому Ада потребует указать квалификацию типа:

```
A : = Numbers1'(1, 2, 3, 4, others => 5);
```

В общем случае, при использовании опции **`others`** совместно с любой из двух нотаций, позиционной или именованой, требуется указывать квалификацию типа.

4.1.5 Отрезки (*array slices*)

Для одномерных массивов Ада предусматривает удобную возможность указания нескольких последовательных компонент массива. Такая последовательность компонент массива называется отрезком массива (*array slice*). В общем случае, отрезок массива может быть задан следующим образом:

```
<имя_массива> (<диапазон_значений_индекса>)
```

Таким образом, для переменной `Calculator_Workspace` типа `Stack`, рассмотренных ранее, можно указать отрезок, содержащий элементы с 5-го по 10-й, следующим образом:

```
Calculator_Workspace (5 .. 10) := (5, 6, 7, 8, 9, 10);
```

В данном примере выполняется присваивание значений элементам массива `Calculator_Workspace`, которые попадают в указанный отрезок, с использованием позиционного агрегата массива.

Приведем еще один простой пример:

```
Calculator_Workspace (25 .. 30) := Calculator_Workspace (5 .. 10);
```

Напомним что использование отрезков допускается только для одномерных массивов.

4.1.6 Массивы-константы

Ада допускает использование массивов-констант. В таких случаях, при описании массива-константы, необходимо инициализировать значения всех его элементов. Такая инициализация, как правило, осуществляется с помощью агрегатов массивов. Например:

```
type Months is (Jan, Feb, Mar, .... , Dec);
subtype Month_Days is Integer range 1..31;
type Month_Length is array (Jan..Dec) of Month_Days;


Days_In_Month : constant Month_Length := (31, 28, 31, 30, ... , 31);
```

4.1.7 Атрибуты массивов

С массивами ассоциируются следующие атрибуты:

```
<имя_массива>'First     -- нижняя граница массива
<имя_массива>'Last      -- верхняя граница массива


<имя_массива>'Length    -- количество элементов в массиве
                  -- <имя_массива>'Last - <имя_массива>'First + 1


<имя_массива>'Range     -- подтип объявленный как
                                  -- <имя_массива>'First..<имя_массива>'Last
```

Эти средства очень полезны для организации перебора элементов массивов.

```
for Count in <имя_массива>'Range loop


   ...

end loop
```

В приведенном выше примере, каждый элемент массива будет гарантированно обработан.

4.2 Многомерные массивы

Ада позволяет использовать многомерные массивы. В качестве простого примера многомерного массива рассмотрим двухмерный массив целых чисел `Square`:

```
Square_Size : constant := 5;
subtype Square_Index  is Integer range 1..Square_Size;
type  Square  is array (Square_Index, Square_Index) of Integer;


Square_Var  : Square := ( others => (others => 0) );
```

Здесь, агрегат, который инициализирует переменную `Square_Var` типа `Square` в нуль, построен как агрегат массива массивов, поэтому требуется двойное использование скобок (опции **`others`** использованы для упрощения примера).

Более сложный пример инициализации этой переменной, использующий агрегат с позиционной нотацией, может иметь следующий вид:

```
----------------- столбцы   1   2   3   4   5


Square_Var  : Square := ( ( 1,  2,  3,  4,  5),     -- строка 1
                    ( 6,  7,  8,  9, 10),    -- строка 2
                    (11, 12, 13, 14, 15),    -- строка 3
                    (16, 17, 18, 19, 20),    -- строка 4
                                    (21, 22, 23, 24, 25) );   -- строка 5
```

Доступ к элементам такого массива можно организовать следующим образом:

```
Square_Var(1, 5) := 5;
Square_Var(5, 5) := 25;
```

Возможно использование альтернативного способа для описания подобного двумерного массива. Его можно описать как массив рядов (иначе - строк), где каждый ряд является одномерным массивом целых чисел `Square_Row`.

```
Square_Size : constant := 5;
subtype Square_Index  is Integer range 1..Square_Size;


type  Square_Row is array (Square_Index) of Integer;
type  Square     is array (Square_Index) of Square_Row;


Square_Var  : Square := ( others => (others => 0) );
```

Примечательно, что инициализация переменных в обоих вариантах реализации двумерного массива выполняется одинаково.

В этом случае, доступ к элементам массива можно организовать следующим образом:

```
Square_Var (1)(5) := 5;
Square_Var (5)(5) := 25;
```

С многомерными массивами можно использовать те же атрибуты, которые допустимы для простых одномерных массивов. При этом несколько изменяется форма указания атрибутов:

```
<имя_массива>'First(N)
<имя_массива>'Last(N)
<имя_массива>'Length(N)
<имя_массива>'Range(N)
```

В данном случае, значение определяемое, например, как `<имя_массива>'Range(N)` будет возвращать диапазон `N`-мерного индекса.

4.3 Типы неограниченных массивов (*unconstrained array*), предопределенный тип `String`

До настоящего момента мы рассматривали только такие массивы у которых диапазон значений индекса был заведомо известен. Такие массивы называют ограниченными массивами, и они могут быть использованы для создания экземпляров объектов с четко определенными во время описания типа границами.

В дополнение к таким типам, Ада позволяет описывать типы массивов, которые не имеют четко определенного диапазона для индексных значений, когда описание типа не определяет границ массива. Поэтому такие массивы называют неограниченными массивами (*unconstrained array*).

Типы неограниченных массивов позволяют описывать массивы, которые во всех отношениях идентичны обычным массивам, за исключением того, что их размер не указан. Ограничение массива (указание диапазона для индексных значений) производится при создании экземпляра объекта такого типа.

Таким образом, описание неограниченого массива предоставляет целый класс массивов, которые содержат элементы одинакового типа, имеют одинаковый тип индекса и одинаковое количество индексов, но при этом разные экземпляры объеков такого типа будут иметь разные размеры.

Этот механизм удобно использовать в случаях когда массив произвольного размера необходимо передать в подпрограмму как параметр.

Примером описания неограниченного массива целых чисел может служить следующее:

```
type  Numbers_Array   is array (Positive range <>) of Integer;
```

Символы "**<>**" указывают, что диапазон значений индекса должен быть указан при описании

объектов типа `Numbers_Array`. Переменная такого типа может быть описана следующим образом:

```
Numbers : Numbers_Array (1..5) := (1, 2, 3, 4, 5);
```

Здесь, при описании переменной `Numbers` предусматривается ограничение (*constraint*) размеров массива - указывается диапазон значений индекса - `(1..5)`.

Пакет *Standard* предоставляет предопределенный тип `String`, который описывается как неограниченный массив символов:

```
type String is array (Positive range <>) of Character;
```

Таким образом, тип `String` может быть использован для описания обширного класса символьных массивов, которые идентичны во всем, кроме количества элементов в массиве.

Также как и в предыдущем примере описания переменной `Numbers`, для создания фактического массива типа `String`, мы должны предусмотреть ограничение диапазона возможных значений индекса:

```
My_Name : String (1..20);
```

Здесь, ограничение диапазона индексных значений находится в диапазоне `1..20`. Преимущество такого подхода в том, что все описанные строки имеют один и тот же тип, и могут, таким образом, использоваться как параметры подпрограмм. Такой дополнительный уровень абстракции позволяет более общее использование подпрограмм обработки строк.

Необходимо заметить, что для инициализации объектов типа `String`, можно использовать агрегаты, поскольку тип `String`, по сути, является массивом символов. Однако, более цивилизованным способом будет использование строковых литералов. Так, вышеописанную переменную `My_Name`, можно инициализировать следующим образом:

```
My_Name := "Alex                ";
```

Следует учесть, что в приведенном примере, строковый литерал, указывающий имя, необходимо дополнить пробелами, чтобы его размер совпадал с размером описанной переменной. В противном случае, компилятор может выдать предупреждение о возбуждении исключения *Constraint_Error* во время выполнения программы.

При описании строк, которым присваиваются начальные значения, границы диапазона можно не указывать:

```
Some_Name   : String := "Vasya Pupkin";
Some_Saying : constant String := "Beer without vodka is money to wind!";
```

Для обработки каждого элемента переменной, которая порождается при использовании типа неограниченного массива, требуется использование таких атрибутов типа массив, как `A'Range`, `A'First` и т.д., поскольку не известно какие индексные значения будет иметь обрабатываемый массив.

```
My_Name    : String (1..20);
My_Surname : String (21..50);
```

Обычно, неограниченные массивы реализуются с объектом который хранит значения границ диапазона индекса и указатель на массив.

4.4 Стандартные операции для массивов

Существует несколько операций, которые могут выполняться не только над отдельными элементами массива, но и над целым массивом.

4.4.1 Присваивание

Целому массиву может присваиваться значение другого массива. Оба массива должны быть одного и того же типа. Если оба массива одного и того же неограниченного типа, то они должны содержать одинаковое количество элементов.

```
declare
    My_Name   : String(1..10)  := "Dale      ";
    Your_Name : String(1..10)  := "Russell   ";
    Her_Name  : String(21..30) := "Liz       ";
    His_Name  : String(1..5)   := "Tim  ";
begin
    Your_Name := My_Name;     -- это корректно, поскольку в обоих случаях
    Your_Name := Her_Name;    -- оба массива имеют одинаковое количество
                              -- элементов
    His_Name  := Your_Name;   -- это приведет к возбуждению исключения:
                              -- хотя обе переменные одного и того же типа,
                              -- но они имеют различную длину (число элементов)
end;
```

4.4.2 Проверки на равенство и на неравенство

Проверки на равенство и на неравенство доступны почти для всех типов Ады. Два массива считаются равными если каждый элемент первого массива равен соответствующему элементу второго массива.

```
if Array1 = Array2 then....
```

4.4.3 Конкатенация

Символ **&** может быть использован как знак операции конкатенации двух массивов.

```
declare
    type Vector is array(Positive range <>) of Integer;

    A : Vector (1..10);
    B : Vector (1..5) := (1, 2, 3, 4, 5);
    C : Vector (1..5) := (6, 7, 8, 9, 10);
begin
    A := B & C;
    Put_Line("hello" & " " & "world");
end;
```

4.4.4 Сравнение массивов

Для сравнения одномерных массивов могут быть использованы следующие знаки операций "**<**", "**<=**", "**>**" и "**>=**". Они наиболее полезны при сравнении массивов символов (строк).

```
"hello" < "world"        -- возвращает результат "истина" (True)
```

В общем случае, можно сравнивать только те массивы у которых можно сравнивать между собой индивидуальные компоненты. Таким образом, например, нельзя сравнивать массивы записей для которых не определена операция сравнения. (то есть, чтобы обеспечить возможность сравнения массивов записей, необходимо, чтобы была определена операция сравнения для записи компонента массива).

4.4.5 Логические операции

Если знаки логических операций "**and**", "**or**", "**xor**", "**not**" допускается использовать для индивидуальных компонентов массива, то использование знаков логических операций для такого массива также будет допустимым (например, в случае массива логических значений типа `Boolean`).

4.5 Динамические массивы

Ада позволяет не указывать размеры массива при написании программы. В этом случае размеры массива не фиксируются во время компиляции программы, а определяются во время ее выполнения, что во многих случаях более предпочтительно. Массивы подобного вида известны как динамические массивы. Кроме того, в отличие от многих других языков программирования, Ада позволяет использование динамических массивов в качестве значения результата, возвращаемого функцией.

```
declare
    X      : Integer := Y    -- значение Y описано где-то в другом месте
    A      : array (1..X) of Integer;
begin
    for I in A'Range loop

        . . .

    end loop;
end;



procedure Demo(Item  : String) is
    Copy   : String(Item'First..Item'Last) := Item;
    Double : String(1..2 * Item'Length) := Item & Item;
begin
       . . .
```

Следует заметить, что не стоит позволять вводу пользователя устанавливать размер массива, и приведенный пример (с декларативным блоком) не должен использоваться как способ решения этой задачи. Использование второго примера наиболее типично.

5. Записи (*record*)

Понятие записи, также как и понятие массива, является механизмом структурирования данных. Однако, в отличие от массива, запись позволяет сгруппировать в одном объекте набор объектов которые могут принадлежать различным типам. При этом объекты из которых состоит запись часто называют компонентами или полями записи.

Для работы с записями, Ада предлагает средства подобные тем, которые предоставляют другие современные языки программирования, а также дополняет их некоторыми своими особенностями. Также как и для массивов, для записей предусматривается использование агрегатов. Использование дискриминантов позволяет создавать вариантные записи, указывать размер для записи переменного размера и выполнять инициализацию компонентов записи.

5.1 Простые записи

5.1.1 Описание простой записи

Как уже было сказано, запись - это структура данных состоящая из набора различных компонентов. В Аде, для описания такой структуры данных, необходимо описать тип записи. В общем случае, описание типа записи имеет следующий вид:

```
type <имя_записи> is
   record
      <имя_поля_1> : <тип_поля_1>;
      <имя_поля_2> : <тип_поля_2>;
         . . .
      <имя_поля_N> : <тип_поля_N>;
      end record;
```

Например:

```
type Bicycle is
   record
      Frame      : Construction;
      Maker      : Manufacturer;
      Front_Brake : Brake_Type;
      Rear_Brake  : Brake_Type;
   end record;

My_Bicycle  : Bicycle;
```

Примечательно, что описание индивидуальных компонентов записи выглядит как описание переменных. Также, следует заметить, что описание типа записи не создает экземпляр объекта записи. В приведенном выше примере, тип `Bicycle` описывает структуру записи, а переменная `My_Bicycle` типа `Bicycle` - является экземпляром записи.

В отличие от массивов, Ада не позволяет создавать анонимные записи. Таким образом, следующий пример описания будет неправильным:

```
My_Bicycle  : record                          -- использование анонимных
                                          -- записей - ЗАПРЕЩЕНО!!!
              Frame      : Construction;
              Maker      : Manufacturer;
              Front_Brake : Brake_Type;
              Rear_Brake  : Brake_Type;
                    end record;
```

Из этого следует, что сначала необходимо описать тип записи, а затем описывать объекты этого типа.

5.1.2 Значения полей записи по-умолчанию

При описании записи, полям записи могут быть назначены значения по-умолчанию. Эти значения будут использоваться всякий раз при создании экземпляра записи данного типа (до тех пор пока они не будут инициализированы другими значениями).

```
type Bicycle is
   record
      Frame      : Construction := CromeMolyebdenum;
      Maker      : Manufacturer;
      Front_Brake : Brake_Type   := Cantilever;
      Rear_Brake  : Brake_Type   := Cantilever;
      end record;
```

5.1.3 Доступ к полям записи

В Аде организация доступа к индивидуальным полям записи осуществляется с помощью точечной нотации, за именем переменной-записи, которое сопровождается точкой, следует имя поля записи. Например:

```
Expensive_Bike  : Bicycle;


Expensive_Bike.Frame        := Aluminium;
Expensive_Bike.Manufacturer := Cannondale;
Expensive_Bike.Front_Brake  := Cantilever;
Expensive_Bike.Rear_Brake   := Cantilever;


if Expensive_Bike.Frame = Aluminium then ...
```

Это идентично организации доступа к полям записи в таких языках программирования как Паскаль или Си.

5.1.4 Агрегаты для записей

Так же как и в случае массива, все множество значений элементов записи может присваиваться с помощью агрегата. При этом агрегат должен предоставлять значения для всех компонентов записи даже в случаях когда некоторые компоненты обеспечены значениями по-умолчанию. Для записей различают агрегаты использующие **позиционную**, **именованную** и **смешанную** нотацию.

Примером использования **позиционного** агрегата может служить следующее:

```
Expensive_Bike := (Aluminium, Cannondale, Cantilever, Cantilever);
```

При позиционной нотации порядок следования присваиваемых значений в агрегате соответствует порядку следования полей в описании типа записи.

Альтернативно позиционной нотации, показанной выше, для присваивания таких же значений полям переменной `Expensive_Bike` можно применить агрегат использующий **именованную** нотацию. В этом случае поля записи могут перечисляться в произвольном порядке:

```
Expensive_Bike := (
                  Rear_Brake   => Cantilever
                  Front_Brake  => Cantilever,
                  Manufacturer => Cannondale,
                  Frame        => Aluminium,
```

```
                        );
```

Для записей допускается смешивать в одном агрегате оба варианта нотации. При этом все позиционные значения должны предшествовать именованным значениям. Такой вариант нотации будет **смешанным**.

Также как и в случае агрегатов для массивов, в агрегатах для записей допускается использование опции **`others`**. При этом, для опции **`others`** должен быть представлен хотя бы один компонент, а в случае когда для опции **`others`** предоставляется более одного компонента, все компоненты должны иметь один и тот же тип.

Агрегаты являются удобным средством указания значений полей при описании переменных и констант:

```
Expensive_Bike :  Bicycle := (Aluminium, Cannondale, Cantilever, Cantilever);
```

Одинаковые значения могут присваиваться разным полям записи при использовании символа '|'.

```
Expensive_Bike := (
                Frame              => Aluminium,
                Manufacturer       => Cannondale,
                Front_Brake | Rear_Brake => Cantilever
                        );
```

В заключение обсуждения применения агрегатов для записей рассмотрим следующий обобщающий пример:

```
type Summary  is
   record
     Field_1 : Boolean;
     Field_2 : Float;
     Field_3 : Integer;
     Field_4 : Integer;
   end record;

Variable_1  : Summary := (True, 10.0, 1, 1);  -- позиционная нотация
Variable_2  : Summary := (                    -- именованная нотация
                     Field_4 => 1
                     Field_3 => 1,
                     Field_2 => 10.0,
                     Field_1 => True
                    );
Variable_2  : Summary := (                    -- смешанная нотация
                     True, 10.0,
                     Field_4 => 1,
                     Field_3 => 1
                    );

------------------------- использование символа '|'
Variable_4  : Summary := (
                     True, 10.0,
                     Field_3|Field_4 => 1
                    );
Variable_5  : Summary := (
                     Field_1 => True,
```

```
                        Field_2 => 10.0,
                        Field_3|Field_4 => 1
                       );

-------------------------- использование others
Variable_6  : Summary := (True, 10.0, others => 1);
Variable_7  : Summary := (
                        Field_1 => True,
                        Field_2 => 10.0,
                        others => 1
                                        );
```

5.1.5 Записи-константы

Записи-константы могут быть созданы также как и обычные переменные. В этом случае значения всех полей записи должны быть инициализированы с помощью агрегата или значений определенных по-умолчанию.

```
My_Bicycle : constant Bicycle := ( Hi_Tensile_Steel,
                                Unknown,
                                Front_Brake => Side_Pull,
                                                        Rear_Brake  => Side_Pull );
```

Присваивать новые значения полям записи-константы или самой записи-константе **нельзя**. Необходимо также заметить, что отдельные поля записи не могут быть описаны как константы.

5.1.6 Лимитированные записи

Тип записи может быть описан как лимитированная запись. В качестве примера, рассмотрим следующие описания:

```
type  Person  is  limited
   record
      Name   : String(1..Max_Chs); -- строка имени
      Height : Height_Cm := 0;    -- рост в сантиметрах
      Sex    : Gender;          -- пол
   end record;

Mike    : Person;
Corrina : Person;
```

В случае, когда тип записи является лимитированной записью, компилятор не позволяет выполнять присваивание и сравнение экземпляров этого типа записи.

```
  ...
Mike    := Corrina;       -- ОШИБКА КОМПИЛЯЦИИ!!!
                        -- для лимитированных записей присваивание запрещено
  ...

if  Corrina = Mike  then  -- ОШИБКА КОМПИЛЯЦИИ!!!
                        -- для лимитированных записей сравнение запрещено
   Put_Line("This is strange");
end if;
```

```
. . .
```

В результате, при компиляции показанного выше кода, будут выдаваться сообщения об ошибке компиляции.

5.2 Вложенные структуры

В качестве компонентов записи можно использовать составные типы. Например, полем записи может быть массив или какая-либо другая запись. Таким образом, Ада предоставляет возможность построения описаний сложных структур данных. Однако, описания таких структур данных обладают некоторыми характерными особенностями на которые необходимо обратить внимание.

5.2.1 Поля типа массив

В случаях когда какой-либо компонент записи необходимо описать как массив необходимо учесть, что такой компонент не может быть указан как анонимный массив. Это означает, что тип массива для такого компонента записи должен быть предварительно описан.

```
type  Illegal is
   record
      Simple_Field_1: Boolean;
      Simple_Field_2: Integer;
      Array_Field   : array (1..10) of Float; -- использование
                                              -- анонимного массива
                                              -- ЗАПРЕЩЕНО!!!
   end record;


type  Some_Array  is array (1..10) of Float;     -- предварительно описанный
                                              -- тип массива
type  Legal is
   record
      Simple_Field_1: Boolean;
      Simple_Field_2: Integer;
      Array_Field   : Some_Array;          -- компонент предварительно
                                              -- описанного типа массив
      end record;
```

Также следует учесть, что в качестве компонентов записей не допускается использование неограниченных массивов. Рассмотрим следующий пример:

```
type  Some_Array  is array (Integer range <>) of Float; -- неограниченный
                                                  -- массив
type  Some_Record is
   record
      Field_1: Boolean;
      Field_2: Integer;
      Field_3: Some_Array (1..10);  -- описание компонента записи
                                    -- задает ограничение индекса
      end record;
```

Здесь, тип `Some_Array` - это неограниченный массив. Поэтому, при описании поля `Field_3` записи `Some_Record` указывается ограничение значений индекса для массива - `(1..10)`. После этого, компонент `Field_3` становится ограниченным массивом.

Для доступа к индивидуальному компоненту поля `Field_3` какой-либо переменной типа `Some_Record` можно использовать:

```
. . .
Some_Var :  Some_Record;
. . .
Some_Var.Field_3(1) := 1;
```

Для инициализации всех значений какой-либо переменной типа `Some_Record` можно использовать агрегаты. В качестве демонстрации, приведем несколько примеров.

```
Some_Var_1 :  Some_Record := (False, 0, (1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
Some_Var_2 :  Some_Record := (
                        Field_1 => False,
                        Field_2 => 0,
                        Field_3 => (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
                       );

Some_Var_3 :  Some_Record := (
                        Field_1 => True,
                        Field_2 => 10,
                        Field_3 => (others => 0)
                                          );
```

Из приведенных примеров видно, что для инициализации простых полей `Field_1` и `Field_2` записей `Some_Var_1`, `Some_Var_2`, `Some_Var_3` типа `Some_Record` используются обычные литералы соответствующего типа, а для инициализации поля `Field_3`, которое является массивом, используется агрегат массива. Таким образом, для инициализации подобных структур необходимо использовать вложенные агрегаты.

5.2.2 Поля записей типа `String`

Частным случаем использования массивов в качестве компонентов записей являются строки `String`. Тип `String`, по сути, является предопределенным неограниченным массивом символов, поэтому для строк, при описании полей записи, допускается как предварительное описание какого-либо строкового типа или подтипа, так и непосредственное использование типа `String`. Например:

```
type    Name_String     is new String(1..10);
subtype Address_String  is String(1..20);


type  Person  is
   record
       First_Name: Name_String;
       Last_Name : Name_String;
       Address   : Address_String;
       Phone     : String(1..15);
       end record;
```

В приведенном выше примере описания типа записи `Person`, для описания полей

First_Name и Last_Name используется самостоятельный строковый тип Name_String, производный от типа String. Для описания поля Address, используется подтип Address_String. Следует заметить, что тип Name_String и подтип Address_String, оба описывают ограниченные строки (или массивы символов). При описании поля Phone непосредственно использован тип String. В этом случае, для типа String, указывается ограничение для значений индекса - (1..15).

Для строковых полей, вместо агрегатов массива допускается использование строковых литералов. Например:

```
Chief : Person := (
                First_Name  => "Matroskin ",
                Last_Name   => "Kot      ",
                Address     => "Prostokvashino     ",
                Phone       => "8-9-222-333    "
                        );
```

5.2.3 Вложенные записи

Бывают случаи, когда компонент записи сам является записью, или, говоря иначе, требуется использование вложенных записей. Например:

```
type  Point is record
   X : Integer;
   Y : Integer;
end record

type Rect is record
   Left_Hight_Corner : Point;
   Right_Low_Corner  : Point;
end record

P : Point := (100, 100);
R : Rect;
```

В этом случае, доступ к полям переменной R типа Rect может быть выполнен следующим образом:

```
R.Left_Hight_Corner.X := 0;
R.Left_Hight_Corner.Y := 0;

R.Right_Low_Corner := P;
```

Для указания всех значений можно использовать агрегаты.

```
R_1 : Rect := ( (0, 0), (100, 100) );
R_2 : Rect := (
            Left_Hight_Corner => (Y => 0, X => 0),
            Right_Low_Corner  => (100, 100)
                    );
```

Как видно из приведенных примеров, здесь используются вложенные агрегаты.

5.3 Дискриминанты

Особенность всех ранее рассмотренных записей в том, что они всегда имеют строго определенные структуры. Действительно, число полей таких записей и их размеры строго зафиксированы на этапе описания типа записи и никак не могут быть изменены впоследствии. Однако, в реальной жизни, достаточно часто возникают ситуации когда необходимо сделать структуру записи зависимой от каких-либо условий.

Для решения такого рода задач, Ада разрешает записям содержать дополнительные поля - дискриминанты. Такие дополнительные поля являются средством "параметризации" и помогают выполнять для записей требуемую настройку. В этом случае, разные экземпляры записей могут принадлежать одному и тому же типу, но при этом иметь разный размер и/или разное количество полей.

Следует заметить, что значения дискриминантов записей должны быть дискретного или ссылочного типа.

5.3.1 Вариантные записи

Использование дискриминантов позволяет конструировать вариантные записи (или, называемые иначе, записи с вариантами). В этом случае, значение дискриминанта определяет наличие или отсутствие некоторых полей записи.

Рассмотрим следующий пример:

```
type Vehicle is (Bicycle, Car, Truck, Scooter);

type Transport (Vehicle_Type : Vehicle) is
   record
      Owner      : String(1..10);     -- владелец
      Description : String(1..10);     -- описание
      case Vehicle_Type is
         when Car   =>
                     Petrol_Consumption : Float; -- расход бензина
         when Truck  =>
                     Diesel_Consumption : Float; -- расход солярки
                     Tare               : Real;  -- вес тары
                     Net                : Real;  -- вес нетто
         when others =>
                     null;
      end case;
   end record;
```

В представленном описании типа записи `Transport`, поле `Vehicle_Type` (vehicle - транспортное средство) является дискриминантом.

В данном случае, значение дискриминанта должно быть указано при описании экземпляра записи. Таким образом, можно создавать различные объекты у которых значения дискриминантов будут отличаться. Заметим также, что при описании таких объектов допускается указывать только значение дискриминанта, а присваивание значений остальным полям записи можно выполнять позже. Агрегаты для записей с дискриминантами должны включать значение дискриминанта как первое значение. Рассмотрим следующие описания:

```
My_Car       : Transport (Car);
```

```
My_Bicycle  : Transport (Vehicle_Type => Bicycle);
His_Car     : Transport := (Car, "dale      ", "escort  ", 30.0);
```

Здесь, переменные `My_Car` и `His_Car`, у которых дискриминант имеет значение `Car`, содержат поля: `Owner`, `Description` и `Petrol_Consumption`. При этом, попытка обращения к таким полям этих переменных как `Tare` и/или `Net` будет не допустима. В результате, это приведет к генерации ошибки ограничения (*constraint error*), что может быть обнаружено и обработано при использовании механизма исключений (*exceptions*) язака Ада.

Следует особо отметить, что несмотря на то, что дискриминант является полем записи, непосредственное присваивание значения дискриминанту запрещено.

Также отметим, что в приведенных примерах тип записи, для простоты, имеет только один дискриминант. Реально, запись может содержать столько дискриминантов, сколько необходимо для решения конкретной задачи.

И в заключение укажем несколько общих особенностей описания записей с вариантами:

- вариантная часть всегда записывается последней
- запись может иметь только одну вариантную часть, однако внутри вариантной части разрешается указывать другой раздел вариантов
- для каждого значения дискриминанта должен быть указан свой список компонентов
- альтернатива **`others`** допустима только для последнего варианта, она задает все оставшиеся значения дискриминанта (возможно, и ни одного), которые не были упомянуты в предыдущих вариантах
- если список компонентов варианта задан как **`null`**, то такой вариант не содержит никаких компонентов

5.3.2 Ограниченные записи (*constrained records*)

Записи `My_Car`, `My_Bicycle` и `His_Car`, которые мы рассматривали выше, называют ограниченными. Для таких записей значение дискриминанта определяется при описании экземпляра (переменной) записи. Таким образом, однажды определенный дискриминант в последствии никогда не может быть изменен. Так, запись `My_Bicycle` не имеет полей `Tare`, `Net`, `Petrol_Consumption`, и т.д. При этом, компилятор Ады даже не будет распределять пространство для этих полей.

Из всего этого следует общее правило: любой экземпляр записи, который описан с указанием значения дискриминанта, будет называться **ограниченным** (*constrained*). Его дискриминант никогда не может принимать значение, отличное от заданного при его описании.

5.3.3 Неограниченные записи (*unconstrained records*)

Ада позволяет описывать экземпляры записей без указания начального значения для дискриминанта, тогда запись будет называться **неограниченной** (*unconstrained*). В этом случае дискриминант может принимать любое значение на протяжении всего времени существования записи, иначе говоря, дискриминант можно изменять. Очевидно, что в этом случае, компилятор должен будет распределить достаточное пространство для того, чтобы иметь возможность разместить наибольшую по размеру запись.

Однако, для выполнения вышесказанного, дискриминант записи обязан иметь значение по-умолчанию, которое указывается при описании типа записи.

```
type Accounts is (Cheque, Savings);
type Account (Account_Type: Accounts := Savings) is
   record
      Account_No  : Positive;
      Title       : String(1..10);
      case Account_Type is
         when Savings  => Interest : Rate;
         when Cheque   => null;
      end case;
      end record;
```

Здесь, дискриминант записи `Account` имеет значение по-умолчанию `Savings`. Теперь, мы можем описать запись:

```
Household_Account : Account;
```

Такая запись будет создана с определенным по-умолчанию значением дискриминанта. Но теперь, мы позже, при необходимости, можем изменить тип этой записи.

```
Household_Account:= (Cheque, 123_456, "household ");
```

В общем следует заметить, что Ада требует чтобы при описании типа записи значения дискриминантов по-умолчанию либо указывались для всех дискриминантов, либо не указывались вовсе. При этом, необходимо учесть, что если тип записи описан с указанием значений дискриминантов по-умолчанию и, затем, при описании экземпляра записи было указано значение дискриминанта, то такой экземпляр записи, в результате, будет ограниченным.

Также необходимо особо отметить, что значение дискриминанта неограниченного объекта запрещается изменять независимо от изменения значений других полей, а непосредственное присваивание значений дискриминанта, как уже говорилось, вовсе запрещено. Поэтому, единственным способом изменения значения дискриминанта является присваивание значения всему объекту. Кроме того, присваивание значений всему объекту сразу является единственным способом изменения значений тех компонентов, у которых определение подтипа зависит от значения дискриминанта. Для пояснения последнего, рассмотрим пример:

```
type  Property  is array (Positive range <>) of Float;
type  Man (Number: Positive := 2; Size: Positive := 10) is
   record
      Name       : String (1..Size);
      Prop_Array  : Property (1..Number);
   end record;

The_Man : Man;

The_Man.Name       := "Ivanov I I";
The_Man.Prop_Array := (25.0, 50.0);

 ...

The_Man := (
          Number    => 3,
          Size      => 8,
          Name      => "Pyle I C",
          Prop_Array => (25.0, 50.0, 160.5)
```

```
                    );
```

Здесь, первоначально объект `The_Man` описан как запись, значения дискриминантов которой устанавливаются по умолчанию. Затем, значения дискриминантов изменяются, но это изменение выполняется согласно требований Ады: осуществляется присваивание значения всей переменной.

5.3.4 Другие использования дискриминантов

Также как и при использовании в качестве селектора выбора варианта в записи, дискриминант может быть использован для спецификации длины массива, который является компонентом записи.

```
type Text (Length  : Positive := 20) is
   record
      Value : String(1..Length);
      end record;
```

В этом случае длина массива зависит от значения дискриминанта. Как указано выше, запись может быть описана как ограниченная (*constrained*) или как неограниченная (*unconstrained*).

Обычно, такие текстовые записи используются как строки переменной длины для текстовой обработки.

Еще одним возможным способом использования дискриминантов могут быть описания различных подтипов какого-либо базового типа. В данном случае смысл идеи состоит в том, что подтипы можно использовать для создания объектов с конкретными значениями дискриминантов. Такой подход удобен при работе с типами, которые содержат большое количество дискриминантов.

6. Подпрограммы

В Аде, также как и во всех современных языках программирования, подпрограммы позволяют программисту группировать инструкции в самостоятельные, логически законченные алгоритмические единицы, которые, в последствии, могут быть вызваны и выполнены в любом месте программы. Они являются элементарным базовым средством для повторного использования кода и разделения одной большой задачи на самостоятельные подзадачи меньшего размера (декомпозиция). Использование подпрограмм позволяет уменьшить как общий размер исходных текстов программы, так и общий размер результирующих исполняемых файлов. Таким образом, применение подпрограмм облегчает общее управление проектом и упрощает его сопровождение.

При этом, подпрограммы Ады обладают некоторыми свойствами, которые будут новыми для программистов использующих Паскаль и/или Си. Совмещение (*overloading*), именованные параметры, значение параметров по-умолчанию, режимы передачи параметров и возврата значений - все это значительно отличает подпрограммы языка Ада.

6.1 Общие сведения о подпрограммах

Также как и в Паскале, подпрограммами Ады являются процедуры и функции.

Подпрограммы могут иметь параметры различных типов или не иметь ни одного параметра.

При описании механизмов передачи параметров, как правило, используются следующие понятия:

формальный параметр	Параметр который используется при описании подпрограммы (процедуры или функции).
фактический параметр	Объект, который передается в подпрограмму при вызове подпрограммы. В качестве фактического параметра подпрограммы может быть передано целое выражение, при этом режимом передачи параметра не может быть режим "**`out`**".

Каждый формальный параметр подпрограммы имеет имя, тип и режим передачи.

Подпрограммы могут содержать локальные описания типов, подтипов, переменных, констант, других подпрограмм и пакетов.

Подпрограмма Ады, как правило, состоит из двух частей: спецификации и тела. Спецификация описывает интерфейс обращения к подпрограмме, другими словами - "что" обеспечивает подпрограмма. Тело подпрограммы описывает детали реализации алгоритма работы подпрограммы, то есть, "как" подпрограмма устроена.

Разделение описания подпрограммы на спецификацию и тело не случайно, и имеет большое значение. Такой подход позволяет предоставить пользователю подпрограммы только ее спецификацию и оградить, и даже избавить его от деталей реализации подпрограммы.

Однажды скомпилированная и сохраненная в атрибутивном файле спецификация может быть проверена на совместимость с другими подпрограммами (и пакетами) когда они будут компилироваться. Таким образом, при проектировании, мы можем представить только спецификацию подпрограммы и передать ее компилятору. При предоставлении большого количества фиктивных подпрограмм-заглушек (*stubs*) мы можем осуществлять предварительное тестирование проекта всей системы и обнаруживать любые ошибки проектирования до того как будет потрачено много усилий на реализацию конкретных решений (идеи данного подхода рассматриваются также при обсуждении концепции пакетов).

Необходимо заметить, что спецификации подпрограмм, как правило, помещаются в спецификации пакетов, а тела подпрограмм - в тела пакетов.

Кроме того, подпрограмма может быть самостоятельным независимым программным модулем. В этом случае, спецификация и тело подпрограммы помещаются в разные файлы (в файл спецификации и в файл тела, соответственно). Следует также заметить, что помещать спецификацию в отдельный файл, когда подпрограмма не является самостоятельной единицей компиляции не обязательно. В этом случае, спецификация подпрограммы может отсутствовать.

6.1.1 Процедуры

Процедуры Ады подобны процедурам Паскаля и используются для реализации самых разнообразных алгоритмов.

Общий вид описания процедуры выглядит следующим образом:

```
procedure имя_процедуры [ (формальные_параметры) ] ;
```

спецификация процедуры, определяющая имя процедуры и профиль ее формальных параметров (если они есть)

Общий вид тела процедуры:

```
procedure имя_процедуры [ (формальные_параметры) ] is
```

спецификация процедуры, определяющая имя процедуры и профиль ее формальных параметров (если они есть)

```
      . . .                                  описательная (или декларативная) часть, которая может содержать
                                             локальные описания типов, переменных, констант, подпрограмм...

begin

      . . .                                  исполняемая часть процедуры, которая описывает алгоритм работы
                                             процедуры;
                                             обязана содержать хотя бы одну инструкцию

end [ имя_процедуры ];
                                             здесь, указание имени процедуры опционально
```

Таким образом, описание содержит только спецификацию процедуры и определяет правила ее вызова (иначе - интерфейс), а тело содержит спецификацию и последовательность инструкций, которые выполняются при вызове процедуры.

Примечательно требование Ады, чтобы исполняемая часть процедуры содержала хотя бы одну инструкцию. Поэтому, как правило на этапе проектирования, при написании процедур-заглушек используется пустая инструкция, например:

```
procedure Demo(X: Integer; Y: Float) is
begin
   null; -- пустая инструкция
end Demo;
```

Вызов процедуры производится также как и в языке Паскаль, например:

```
Demo(4, 5.0);
```

Необходимо также заметить, что Ада предоставляет программисту возможность, при необходимости, помещать в любых местах внутри исполнительной части процедуры инструкцию возврата из процедуры - **`return`**.

6.1.2 Функции

Функции во многом подобны процедурам, за исключением того, что они возвращают значение в вызвавшую их подпрограмму. Также можно сказать, что функция - это подпрограмма которая осуществляет преобразование одного или нескольких входных значений в одно выходное значение.

Общий вид описания функции выглядит следующим образом:

```
function имя_функции [ (формальные_параметры) ]
      return тип_возвращаемого_значения ;
                                             спецификация функции, определяющая имя функции, профиль ее формальных
                                             параметров (если они есть) и тип возвращаемого значения
```

Общий вид тела функции:

```
function имя_функции [ (формальные_параметры) ]
      return тип_возвращаемого_значения is
                                             спецификация функции, определяющая имя функции, профиль ее формальных
                                             параметров (если они есть) и тип возвращаемого значения

      . . .                                  описательная (или декларативная) часть, которая может содержать локальные
                                             описания типов, переменных, констант, подпрограмм...
```

```
begin

      . . .                                  исполнительная часть функции, которая описывает алгоритм работы функции;
                                             обязана содержать хотя бы одну инструкцию возврата значения - return

end [ имя функции ];
                                             здесь, указание имени функции опционально
```

Использование инструкции возврата значения - **`return`** очень похоже на то, что используется в языке Си, при этом, функция может иметь сколько угодно инструкций возврата значения.

Функция может быть вызвана как часть выражения в инструкции присваивания или как аргумент соответствующего типа при вызове другой функции или процедуры. Другими словами - функция, возвращающая значения заданного типа, может быть использована везде, где может быть использована переменная этого типа.

Хотя режимы передачи параметров в подпрограммы будут подробно рассмотрены несколько позже, здесь, необходимо сделать несколько важных замечаний, которые имеют значение для функций Ады.

Согласно традиций стандарта Ada83, для передачи параметров в функцию разрешается использовать только режим "**`in`**". Поэтому, функция, через свои параметры, может только импортировать данные из среды вызвавшей эту функцию. При этом, параметры функции не могут быть использованы для изменения значений переменных в среде вызвавшей функцию. Таким образом, в отличие от традиций языка Си, функции Ады не обладают побочными эффектами.

Стандарт Ada95 ввел новый режим для передачи параметров - **`access`**. Этот режим разрешается использовать для передачи параметров в функции. Следует заметить, что использование этого режима допускает написание функций обладающих побочными эффектами.

6.1.3 Локальные переменные

Как уже говорилось, подпрограммы (и процедуры, и функции) могут содержать локальные переменные. Такие переменные доступны только внутри подпрограммы и не видимы извне этой подпрограммы. Время жизни (время существования) таких переменных определяется временем жизни (временем выполнения) подпрограммы.

Во время работы программы, при входе в подпрограмму, имеющую локальные переменные, в стеке времени выполнения происходит автоматическое распределение пространства для локальных переменных данной подпрограммы. При выходе из подпрограммы пространство стека времени выполнения, распределенное для локальных переменных данной подпрограммы, автоматически возвращается системе.

6.1.4 Локальные подпрограммы

Также как и Паскаль, и в отличие от Си, Ада позволяет встраивать одни подпрограммы в другие подпрограммы, конструируя один общий компилируемый модуль. Другими словами, подпрограмма Ады может содержать внутри себя вложенные подпрограммы, которые не будут видимы извне этой подпрограммы. К таким локальным подпрограммам можно обращаться только из подпрограммы которая их содержит.

```
with Ada.Text_IO;          use Ada.Text_IO;
with Ada.Integer_Text_IO;  use Ada.Integer_Text_IO;
```

```
procedure Ive_Got_A_Procedure is

    X : Integer := 6;
    Y : Integer := 5;

    procedure Display_Values (Number : Integer) is
    begin
        Put (Number);
        New_Line;
    end Display_Values;

begin
    Display_Values (X);
    Display_Values (Y);
end Ive_Got_A_Procedure;
```

В этом примере область видимости процедуры `Display_Values` ограничивается процедурой `Ive_Got_A_Procedure`. Таким образом, процедура `Display_Values` "не видна" и не может быть вызвана из любого другого места.

6.1.5 Раздельная компиляция

В предыдущем примере, если будет произведено какое-нибудь изменение кода, то обе процедуры должны быть переданы компилятору (поскольку обе находятся в одном файле с исходным текстом). Мы можем разделить эти две компоненты, и поместить их в отдельные файлы, оставляя без изменения ограничения области видимости для процедуры `Display_Values`. Это несколько похоже на директиву `#include`, используемую в языке Си, но, в языке Ада, теперь оба файла становятся независимыми компилируемыми модулями.

В первом файле:

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Ive_Got_A_Procedure is

    X : Integer := 6;
    Y : Integer := 5;

    procedure Display_Values(Number : Integer) is separate;

begin
    Display_Values(X);
    Display_Values(Y);
end Ive_Got_A_Procedure;
```

Во втором файле:

```
separate(Ive_Got_A_Procedure)  -- примечание! нет завершайщего символа
                               -- точки с запятой

procedure Display_Values(Number : Integer) is
begin
```

```
    Put(Number);

    New_Line;
end Display_Values;
```

Выделенный в самостоятельный файл (и ставший отдельно компилируемым модулем), код - идентичен тому, что было в предыдущей версии. Однако теперь, если будет изменена только внутренняя подпрограмма, то только она должна быть подвергнута перекомпиляции компилятором. Это также позволяет разделить программу на несколько частей, что может облегчить ее понимание.

6.1.6 Подпрограммы как библиотечные модули

Любая подпрограмма Ады, при необходимости, может быть оформлена как абсолютно самостоятельный независимый библиотечный подпрограммный модуль.

Рассмотрим как это делается на примере процедур `Ive_Got_A_Procedure` и `Display_Values`, из предыдущего примера о раздельной компиляции. Теперь, процедура `Display_Values` будет оформлена как самостоятельный библиотечный подпрограммный модуль, а процедура `Ive_Got_A_Procedure` будет ее использовать.

В этом случае, полное описание процедуры `Display_Values` будет состоять из двух файлов: файла спецификации и файла тела процедуры.

> **Примечание:**
> В системе компилятора *GNAT* существует соглашение согласно которому файлы спецификаций имеют расширение `ads` (*ADa Specification*), а файлы тел имеют расширение `adb` (*ADa Body*).

Файл спецификации процедуры `Display_Values` (`display_values.ads`) будет иметь следующий вид:

```
procedure Display_Values(Number : Integer);
```

Файл тела процедуры `Display_Values` (`display_values.adb`) будет иметь следующий вид:

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;


procedure Display_Values(Number : Integer) is
begin
    Put(Number);

    New_Line;
end Display_Values;
```

Третий файл - это файл тела процедуры `Ive_Got_A_Procedure` (`ive_got_a_procedure.adb`). В этом случае он будет иметь следующий вид:

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Display_Values;


procedure Ive_Got_A_Procedure is


    X : Integer := 6;

    Y : Integer := 5;
```

```
begin
    Display_Values(X);
    Display_Values(Y);
end Ive_Got_A_Procedure;
```

Примечательно, что теперь, в файле тела процедуры `Ive_Got_A_Procedure`, процедуру `Display_Values`, которая оформлена как самостоятельный библиотечный модуль, необходимо указать в спецификаторе совместности контекста **`with`**.

Также, необходимо заметить, что подпрограммы, оформленные как самостоятельные библиотечные модули, не указываются в спецификаторе использования контекста **`use`**.

6.2 Режимы передачи параметров

Стандарт Ada83 предусматривал три режима передачи параметров для подпрограмм:

"in"
"in out"
"out"

Стандарт Ada95 добавил еще один режим передачи параметров:

access

Все эти режимы не имеют непосредственных аналогов в других языках программирования. Необходимо также отметить следующее:

по-умолчанию, для передачи параметров подпрограммы, всегда устанавливается режим - **"in"** !!!

Для **"in"** / **"out"** скалярных значений используется механизм передачи параметров по копированию-**"in"** (*copy-in*), по копированию-**"out"** (*copy-out*). Стандарт специфицирует, что любые другие типы могут быть переданы по *copy-in*/*copy-out*, или по ссылке.

Ada95 указывает, что лимитированные приватные типы (*limited private types*), которые рассматриваются позднее, передаются по ссылке, для предотвращения проблем нарушения приватности.

6.2.1 Режим "in"

Параметры передаваемые в этом режиме подобны параметрам передаваемым по значению в языке Паскаль, и обычным параметрам языка Си, с тем исключением, что им не могут присваиваться значания внутри подпрограммы.

Это значит, что при входе в подпрограмму, формальный параметр инициализируется значением фактического параметра, при этом, внутри подпрограммы, он является константой и разрешает только чтение значения ассоциированного фактического параметра.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Demo(X : in Integer; Y : in Integer) is
begin
    X := 5;   -- недопустимо, in параметр доступен только по чтению
    Put(Y);
    Get(Y);   -- также недопустимо
```

```
end Demo;
```

Режим "**in**" разрешается использовать и в процедурах, и в функциях.

6.2.2 Режим "in out"

Этот режим непосредственно соответствует параметрам передаваемым по ссылке (подобно *var*-параметрам языка Паскаль).

Таким образом, при входе в подпрограмму, формальный параметр инициализируется значением фактического параметра. Внутри подпрограммы, формальный параметр, использующий этот режим, может быть использован как в левой, так и в правой части инструкций присваивания (другими словами: формальный параметр доступен как для чтения, так и для записи). При этом, если формальному параметру внутри подпрограммы произведено присваивание нового значения, то после выхода из подпрограммы значение фактического параметра заменяется на новое значение формального параметра.

```
procedure Demo(X : in out Integer;
               Y : in     Integer) is

   Z : constant Integer := X;

begin
   X := Y * Z; -- это допустимо!
end Demo;
```

Режим "**in out**" разрешается использовать только в процедурах.

6.2.3 Режим "out"

В этом режиме, при входе в подпрограмму, формальный параметр **не** инициализируется (**!!!**) значением фактического параметра. Согласно стандарта Ada95, внутри подпрограммы, формальный параметр, использующий этот режим, может быть использован как в левой, так и в правой части инструкций присваивания (другими словами: формальный параметр доступен как для чтения, так и для записи). Согласно стандарта Ada83, внутри подпрограммы, такой формальный параметр может быть использован только в левой части инструкций присваивания (другими словами: доступен только для записи). При этом, после выхода из подпрограммы, значение фактического параметра заменяется на значение формального параметра.

```
procedure Demo(X : out Integer;
               Y : in Integer) is

   -- при входе в подпрограмму X не инициализирован!!!

begin
   X := Y;
end Demo;
```

Режим "**out**" разрешается использовать только в процедурах.

6.2.4 Режим access

Поскольку значения ссылочного типа (указатели) часто используются в качестве параметров

передаваемых подпрограммам, Ада предусматривает режим передачи параметров **access**, который специально предназначен для передачи параметров ссылочного типа. Заметим, что подробному обсуждению ссылочных типов Ады далее посвящена самостоятельная глава - *"Ссылочные типы (указатели)"*. Необходимо также обратить внимание на то, что режим передачи параметров **access** был введен стандартом Ada95 и он отсутствует в стандарте Ada83.

При использовании режима **access**, фактический параметр, который предоставляется при вызове подпрограммы, - это любое значение ссылочного типа, которое ссылается (указывает) на объект соответствующего типа. При входе в подпрограмму, формальный параметр инициализируется значением фактического параметра, при этом, Ада производит автоматическую проверку того, что значение параметра не равно **null**. В случае когда значение параметра равно **null** генерируется исключительная ситуация *Constraint_Error* (проще говоря, - ошибка). Внутри подпрограммы, формальный параметр, использующий режим **access**, является константой ссылочного типа и ему нельзя присваивать новое значение, поэтому такие формальные параметры несколько подобны формальным параметрам, использующим режим "**in**". Однако, поскольку параметр является значением ссылочного типа (указателем), то подпрограмма может изменить содержимое объекта на который данный параметр ссылается (указывает). Кроме того, внутри подпрограммы такой параметр принадлежит анонимному ссылочному типу, и поскольку у нас нет возможности определить имя этого ссылочного типа, то мы не можем описать ни одного дополнительного объекта этого типа. Любая попытка конвертирования значения такого параметра в значение именованого ссылочного типа будет проверяться на соответствие правилам области действия для ссылочных типов. При обнаружении нарушения этих правил генерируется исключительная ситуация *Programm_Error*.

```
   ...

function Demo_Access(A : access Integer) return Integer is
begin
   return A.all;
end Demo_Access;


   ...

type  Integer_Access  is access Integer;

Integer_Access_Var  : Integer_Access := new Integer'(1);
Aliased_Integer_Var : aliased Integer;


   ...

X : Integer := Demo_Access(Integer_Access_Var);
Y : Integer := Demo_Access(Aliased_Integer_Var'Access);
Z : Integer := Demo_Access(new Integer);


      . . .
```

Режим **access** разрешается использовать и в процедурах, и в функциях.

При этом необходимо обратить внимание на то, что функции, использующие этот режим для передачи параметров, способны изменять состояние объектов на которые такие параметры ссылаются. То есть, такие функции могут обладать побочными эффектами.

6.3 Сопоставление формальных и фактических параметров

6.3.1 Позиционное сопоставление

Позиционное сопоставление формальных и фактических параметров при вызове подпрограммы достаточно традиционно, и используется во многих языках программирования. При таком сопоставлении, ассоциирование между формальными и фактическими параметрами производится один к одному позиционно, т.е. первый формальный параметр ассоциируется с первым фактическим параметром и т.д.

```
procedure Demo(X : Integer; Y : Integer);  -- спецификация процедуры


 ...

Demo(1, 2);
```

В этом случае, фактический параметр `1` будет подставлен вместо первого формального параметра `X`, а фактический параметр `2` будет подставлен вместо второго формального параметра `Y`.

6.3.2 Именованное сопоставление

Для улучшения читабельности вызовов подпрограмм (а Ада разрабатывалась с учетом хорошей читабельности) Ада позволяет использовать именованное сопоставление формальных и фактических параметров. В этом случае мы можем ассоциировать имя формального параметра с фактическим параметром. Это свойство делает вызовы подпрограмм более читабельными.

```
procedure Demo(X : Integer; Y : Integer);  -- спецификация процедуры


 ...

Demo(X => 5, Y => 3 * 45);   -- именованное сопоставление
                      -- формальных и фактических
                                          -- параметров при вызове
```

Расположение списка параметров вертикально, также способствует улучшению читабельности.

```
Demo(X => 5,
       Y => 3 * 45);
```

Поскольку при именованом сопоставлении производится явное ассоциирование между формальными и фактическими параметрами (вместо неявного ассоциирования, используемого в случае позиционного сопоставления), то нет необходимости строго придерживаться того же самого порядка следования параметров, который указан в спецификации подпрограммы.

```
Demo(Y => 3 * 45,    -- при именованом сопоставлении
    X => 5);      -- порядок следования параметров
                               -- не имеет значения
```

6.3.3 Смешивание позиционного и именованного сопоставления

Ада позволяет смешивать позиционное и именованное сопоставление параметров. В этом случае должно соблюдаться следующее условие: позиционно-ассоциированные параметры

должны предшествовать параметрам, которые ассоциируются по имени.

```
procedure Square(Result :    out Integer;
                 Number : in    Integer) is
begin
   Result := Number * Number;
end Square;
```

В результате, показанная выше процедура `Square` может быть вызвана следующими способами:

```
Square(X, 4);
Square(X, Number => 4);
Square(Result => X, Number => 4);
Square(Number => 4, Result => x);

Square(Number => 4, X);  -- недопустимо, поскольку позиционно-ассоциируемый
                         -- параметр следует за параметром, ассоциируемым
                                      -- по имени
```

6.4 Указание значения параметра по-умолчанию

Для любых "**`in`**"-параметров ("**`in`**" или "**`in out`**"), в спецификации подпрограммы можно указать значение параметра по-умолчанию. Синтаксис установки значения параметра по-умолчанию подобен синтаксису определения инициализированных переменных и имеет следующий вид:

```
with Ada.Text_IO;        use Ada.Text_IO;

procedure Print_Lines(No_Of_Lines: Integer := 1) is

begin
   for Count in 1 .. No_Of_Lines loop
      New_Line;
   end loop;
end Print_Lines;
```

Такое описание устанавливает значение параметра `No_Of_Lines` для случаев когда процедура `Print_Lines` вызывается без указания значения этого параметра (позиционного или именованного).

Таким образом, вызов этой процедуры может иметь вид:

```
Print_Lines;        -- это печатает одну строку
Print_Lines(6);     -- переопределяет значение параметра
                            -- установленное по-умолчанию
```

Подобно этому, если процедура `Write_Lines` была описана как:

```
with Ada.Text_IO;        use Ada.Text_IO;

procedure Write_Lines(Letter      : in Char := '*';
                      No_Of_Lines : in Integer := 1) is

begin
   for I in 1 .. No_Of_Lines loop
      for I in 1 .. 80 loop
         Put(Letter);
```

```
            end loop;
            New_Line;
        end loop;
    end Write_Lines;
```

то она может быть вызвана следующими способами:

```
    Write_Lines;                    -- для параметров Letter и No_Of_Lines
                                    -- используются значения устанавливаемые
                                    -- по-умолчанию
    Write_Lines('-');               -- значение по-умолчанию - для No_Of_Lines
    Write_Lines(no_of_lines => 5);  -- значение по-умолчанию - для Letter
    Write_Lines('-', 5)             -- оба параметра определены
```

6.5 Совмещение (*overloading*)

Поиск новых имен для подпрограмм, которые выполняют одинаковые действия, но с переменными разных типов, всегда является большой проблемой при разработке программного обеспечения. Хорошим примером для иллюстрации такой проблемы является процедура `Insert`.

В подобных случаях, для облегчения жизни программистам, Ада позволяет разным подпрограммам иметь одинаковые имена, предоставляя механизм который называется совмещением (*overloading*).

6.5.1 Совмещение подпрограмм (*subprogram overloading*)

Предоставляя механизм совмещения имен подпрограмм, Ада выдвигает единственное требование: подпрограммы должны быть распознаваемы (или различимы). Две подпрограммы, имеющие одинаковые имена, будут распознаваемы если они имеют разный "профиль". Профиль подпрограммы характеризуется количеством параметров и их типами, а также, если подпрограмма является функцией, - типом возвращаемого значения.

Таким образом, пока компилятор может однозначно определить к какой подпрограмме осуществляется вызов, анализируя совпадение профиля вызываемой подпрограммы со спецификациями представленных подпрограмм, - все будет в порядке. В противном случае, вы получите сообщение об ошибке, указывающее на двусмысленность обращения.

```
    procedure Insert(Item : Integer);  -- две процедуры с одинаковыми именами,
    procedure Insert(Item : Float);    -- но имеющие разный "профиль"
```

Примерами совмещенных подпрограмм могут служить процедуры `Put` и `Get` из стандартного пакета *Ada.Text_IO*.

6.5.2 Совмещение знаков операций (*operator overloading*)

В языках подобных Паскалю знак операции "+" - совмещен. Действительно, он может использоваться для сложения целых и вещественных чисел, и даже строк. Таким образом, очевидно что этот знак операции используется для представления кода который выполняет абсолютно разные действия.

Ада разрешает программистам выполнять совмещение предопределенных знаков операций с их собственным кодом. Следует заметить, в Аде, действие выполняемое знаком операции, реализуется путем вызова функции именем которой является знак операции заключенный в

двойные кавычки. При этом, для обеспечения корректной работы механизма совмещения знаков операций, функция, которая реализует действие знака операции, должна соответствовать обычному требованию механизма совмещения подпрограмм Ады: она должна быть различима, то есть, ее профиль должен быть уникальным. В некоторых случаях, проблему двусмысленности знака операции можно преодолеть непосредственно специфицируя имя пакета (рассматривается далее).

Кроме того, к функциям которые реализуют действия знаков операций предъявляется дополнительное требование: они не могут быть выделены в самостоятельно компилируемые модули, а должны содержаться в другом модуле, таком как процедура, функция или пакет.

Рассмотрим простой пример в котором мы хотим предусмотреть возможность сложения двух векторов:

```
procedure Add_Demo is

   type Vector is array (Positive range <>) of Integer;
   A : Vector(1..5);
   B : Vector(1..5);
   C : Vector(1..5);

   function "+"(Left, Right : Vector) return Vector is

      Result   : Vector(Left'First..Left'Last);
      Offset   : constant Natural := Right'First - 1;

   begin
      if Left'Length /= Right'Length then
         raise Program_Error;       -- исключение,
                                    -- рассматриваются позже
      end if;

      for I in Left'Range loop
         Result(I) := Left(I) + Right(I - Offset);
      end loop;
      return Result;
   end "+";

begin
   A := (1, 2, 3, 4, 5);
   B := (1, 2, 3, 4, 5);
   C := A + B;
end Add_Demo;
```

В этом примере хорошо продемонстрированы многие ранее рассмотренные средства которые характерны для языка программирования Ада.

6.5.3 Спецификатор "use type"

В случае когда какой-либо пакет содержит описания знаков операций, может быть использована несколько модифицированная форма спецификатора использования **use**, которая позволяет использовать описанные в этом пакете знаки операций без необходимости указания имени

пакета в качестве префикса. При этом, в случае использования других компонентов, описанных в этом пакете, необходимость указания имени пакета в качестве префикса сохраняется. Эта модифицированная форма спецификатора использования имеет следующий вид:

```
use type имя_типа
```

Здесь, *имя_типа* указывает тип данных для которого знаки операций будут использоваться без указания имени пакета в качестве префикса.

7. Пакеты

Хотя корни Ады лежат в языке Паскаль, концепция пакетов была заимствована из других языков программирования и подверглась значительному влиянию последних наработок в области разработки программного обеспечения обсуждавшихся в 1970-х годах. Несомненно, что одной из главных инноваций в этот период была концепция программного модуля (или пакета).

Следует заметить, что концепция пакетов не рассматривает то, как программа будет выполняться. Идея данного подхода посвящена проблеме построения программного комплекса, облегчению понимания того как он устроен и, следовательно, упрощению его сопровождения.

7.1 Общие сведения о пакетах Ады

7.1.1 Идеология концепции пакетов

Прежде чем непосредственно рассматривать примеры языковых конструкций необходимо разобраться в чем заключается и что предлагает концепция пакетов.

Пакет - это средство, которое позволяет сгруппировать логически связанные вычислительные ресурсы и выделить их в единый самостоятельный программный модуль. Под вычислительными ресурсами в этом случае подразумеваются данные (типы данных, переменные, константы...) и подпрограммы которые манипулируют этими данными. Характерной особенностью данного подхода является разделение самого пакета на две части: спецификацию пакета и тело пакета. Причем, спецификацию имеет каждый пакет, а тело могут иметь не все пакеты.

Спецификация определяет интерфейс к вычислительным ресурсам (сервисам) пакета доступным для использования во внешней, по отношению к пакету, среде. Другими словами - спецификация показывает "что" доступно при использовании этого пакета.

Тело является приватной частью пакета и скрывает в себе все детали реализации предоставляемых для внешней среды ресурсов, то есть, тело хранит информацию о том "как" эти ресурсы устроены.

Необходимо заметить, что разбиение пакета на спецификацию и тело не случайно, и имеет очень важное значение. Это дает возможность по-разному взглянуть на пакет. Действительно, для использования ресурсов пакета достаточно знать только его спецификацию, в ней содержится вся необходимая информация о том как использовать ресурсы пакета. Необходимость в теле пакета возникает только тогда, когда нужно узнать или изменить реализацию чего-либо внутри самого пакета.

Средства построения такой конструкции как пакет дают программисту мощный и удобный инструмент абстракции данных, который позволяет объединить и выделить в логически законченное единое целое данные и код который манипулирует этими данными. При этом, пакет позволяет программисту скрыть все детали реализации сервисов за развитым функциональным

интерфейсом. В результате, структурное представление программного комплекса в виде набора взаимодействующих между собой компонентов облегчает понимание работы комплекса в целом и, следовательно, позволяет облегчить его разработку и сопровождение.

Необходимо также заметить, что на этапе начального проектирования системы можно предоставлять компилятору только спецификации, обеспечивая детали реализации только самых необходимых элементов. Таким образом проверка корректности общей структуры проекта осуществляется на ранней стадии, когда не потрачено много усилий на разработку реализации отдельных элементов, которые позже придется переделывать (что, к великому сожалению, в реальной жизни происходит достаточно часто).

> **Примечание:**
> В системе компилятора *GNAT* существует соглашение согласно которому файлы спецификаций имеют расширение `ads` (*ADa Specification*), а файлы тел имеют расширение `adb` (*ADa Body*).

7.1.2 Спецификация пакета

Как уже говорилось, спецификация пакета Ады определяет интерфейс доступа к вычислительным ресурсам (сервисам) пакета. Она может содержать описания типов, переменных, констант, спецификации подпрограмм, других пакетов, - все то, что должно быть доступно тому, кто будет использовать данный пакет. Простой пример спецификации пакета может иметь следующий вид:

```
package Odd_Demo is

   type A_String is array (Positive range <>) of Character;

   Pi  : constant Float := 3.14;

   X   : Integer;

   type A_Record is
      record
        Left  : Boolean;
        Right : Boolean;
      end record;

   -- примечательно, что дальше, для двух подпрограмм представлены только
   -- их спецификации, тела этих подпрограмм будут находиться в теле пакета

   procedure Insert(Item : in Integer; Success : out Boolean);
   function Is_Present(Item : in Integer) return Boolean;

end Odd_Demo;
```

Мы можем получить доступ к этим сервисам в нашем коде путем указания данного пакета в спецификаторе совместности контекста **`with`**, а затем использовать полную точечную нотацию. Полная точечная нотация, в общем случае, имеет следующий вид:

```
имя_пакета.имя_используемого_ресурса
```

где *имя_используемого_ресурса* - это имя типа, подпрограммы, переменной и т.д.

Для демонстрации сказанного приведем схематический пример процедуры которая использует показанную выше спецификацию:

```
with Odd_Demo;

procedure Odder_Demo is


    My_Name : Odd_Demo.A_String;
    Radius  : Float;
    Success : Boolean;

begin
    Radius := 3.0 * Odd_Demo.Pi;
    Odd_Demo.Insert(4, Success);
    if Odd_Demo.Is_Present(34) then ...
       . . .
end Odder_Demo;
```

Не трудно заметить, что доступ к ресурсам пакета, текстуально подобен доступу к полям записи.

В случаях, когда использование полной точечной нотации для доступа к ресурсам пакета обременительно, можно использовать инструкцию спецификатора использования контекста **use**. Это позволяет обращаться к ресурсам которые предоставляет данный пакет без использования полной точечной нотации, так, как будто они описаны непосредственно в этом же коде.

```
with Odd_Demo;         use Odd_Demo;

procedure Odder_Demo is


    My_Name : A_String;
    Radius  : Float;
    Success : Boolean;

begin
    Radius := 3.0 * Pi;
    Insert(4, Success);
    if Is_Present(34) then ...
       . . .
end Odder_Demo;
```

Если два пакета, указаны в инструкциях **with** и **use** в одном компилируемом модуле (например, в подпрограмме или другом пакете), то возможно возникновение коллизии имен используемых ресурсов, которые предоставляются двумя разными пакетами. В этом случае можно избавиться от двусмысленности путем возвращения к использованию полной точечной нотации для соответствующих ресурсов.

```
-------------------------------
package No1 is
   A, B, C : Integer;
end No1;

-------------------------------
package No2 is
```

```
    C, D, E : Integer;
end No2;


-------------------------------
with No1;         use No1;
with No2;         use No2;


procedure Clash_Demo is
begin
    A := 1;
    B := 2;


    C := 3;       -- двусмысленность, мы ссылаемся
                  -- на No1.c или на No2.c?


    No1.C := 3;   -- избавление от двусмысленности путем возврата
    No2.C := 3;   -- к полной точечной нотации
end Clash_Demo;
```

Может возникнуть другая проблема - когда локально определенный ресурс "затеняет" ресурс пакета указанного в инструкции **use**. В этом случае также можно избавиться от двусмысленности путем использования полной точечной нотации.

```
package No1 is
    A : Integer;
end No1;


with No1;         use No1;
procedure P is
    A : Integer;
begin
    A := 4;       -- это - двусмысленно
    P.A := 4;     -- удаление двусмысленности путем указания
                  -- имени процедуры в точечной нотации
    No1.A := 5;   -- точечная нотация для пакета
end P;
```

7.1.3 Тело пакета

Тело пакета содержит все детали реализации сервисов, указаных в спецификации пакета. Схематическим примером тела пакета, для показанной выше спецификации, может служить:

```
package body Odd_Demo is


    type List is array (1..10) of Integer;
    Storage_List : List;
    Upto         : Integer;


    procedure Insert(Item    : in  Integer;
                     Success : out Boolean) is
    begin
        . . .
    end Insert;


    function Is_Present(Item : in Integer) return Boolean is
```

```
    begin
        . . .
    end Is_Present;


begin                   -- действия по инициализации пакета
                        -- это выполняется до запуска основной программы!
    for I in Storage_List'Range loop
        Storage_List(I) := 0;
    end loop;
    Upto := 0;
end Odd_Demo;
```

Все ресурсы, указанные в спецификации пакета, будут непосредственно доступны в теле пакета без использования дополнительных инструкций спецификации контекста **with** и/или **use**.

Необходимо заметить, что тело пакета, также как и спецификация пакета, может содержать описания типов, переменных, подпрограмм и т.д. При этом, ресурсы, описанные в теле пакета, не доступны для использования в другом самостоятельном модуле (пакете или подпрограмме). Любая попытка обращения к ним из другого модуля будет приводить к ошибке компиляции.

Переменные, описанные в теле пакета, сохраняют свои значения между успешными вызовами публично доступных подпрограмм пакета. Таким образом, мы можем создавать пакеты, которые сохраняют информацию для более позднего использования (другими словами: сохранять информацию о состоянии).

Раздел "**begin** ... **end**", в конце тела пакета, содержит перечень инструкций инициализации для этого пакета. Инициализация пакета выполняется до запуска на выполнение главной подпрограммы. Это справедливо для всех пакетов. Следует заметить, что стандарт не определяет порядок выполнения инициализации различных пакетов.

7.2 Средства сокрытия деталей реализации внутреннего представления данных

Как уже указывалось, спецификация пакета определяет интерфейс, а его тело скрывает реализацию сервисов предоставляемых пакетом. Однако, в примерах, показанных ранее, от пользователей пакетов были скрыты только детали реализации подпрограмм, а все типы данных были открыто описаны в спецификации пакета. Следовательно, все детали реализации представления внутренних структур данных "видимы" пользователям. В результате, пользователи таких пакетов, полагаясь на открытость представления внутренних структур данных и используя эти сведения, попадают в зависимость от деталей реализации структур данных. Это значит, что в случае какого-либо изменения во внутреннем представлении данных, как минимум, возникает необходимость в полной перекомпиляции всех программных модулей которые используют такую информацию. В худшем случае, зависимые модули придется не только перекомпилировать, но и переделать.

На первый взгляд, сама проблема выглядит достаточно безобидно, а идея скрыть реализацию внутреннего представления данных кажется попыткой ущемления здорового любопытства пользователя. Справедливо заметить, что такие мысли, как правило, возникают при виде простых структур данных представленных в учебных примерах. К сожалению, в реальной

жизни все сложнее. Представьте себе последствия изменения внутреннего представления данных в реальном проекте когда зависимых модулей много и разрабатываются эти модули разными программистами.

Ада позволяет "закрыть" детали внутреннего представления данных и, таким образом, избавиться от проблемы массовых переделок. Такие возможности обеспечивает использование приватных типов (*private types*) и лимитированных приватных типов (*limited private types*).

7.2.1 Приватные типы (*private types*)

Рассмотрим пример пакета который управляет счетами в бухгалтерской книге. При этом, нам необходимо осуществлять полный контроль над всеми манипуляциями которые выполняются с объектами, и мы обеспечиваем пользователям пакета только следующие возможности:

- изъятие средств (`Withdraw`)
- размещение средств (`Deposit`)
- создание счета (`Create`)

Никакие другие пакеты не должны иметь представления о деталях реализации внутренней структуры объекта бухгалтерского счета (`Account`) и, следовательно, иметь к ним доступ.

Для того чтобы выполнить поставленную задачу, мы описываем объект бухгалтерского счета `Account` как приватный тип:

```
package Accounts is
    type Account is private;       -- описание будет представлено позже

    procedure Withdraw(An_Account : in out Account;
                       Amount     : in     Money);

    procedure Deposit( An_Account : in out Account;
                       Amount     : in     Money);
    function Create( Initial_Balance : Money) return Account;
    function Balance( An_Account : in    Account) return Integer;

private                    -- эта часть спецификации пакета
                           -- содержит полные описания
    type Account is
        record
            Account_No : Positive;
            Balance    : Integer;
        end record;
end Accounts;
```

В результате такого описания, тип `Account` будет приватным. Следут заметить, что Ада разрешает использовать следующие предопределенные операции над объектами приватного типа вне этого пакета:

- присваивание
- проверка на равенство (не равенство)
- проверки принадлежности ("**`in`**", "**`not in`**")

Кроме предопределенных операций, над объектами приватного типа, вне этого пакета,

разрешается использовать операции, которые объявлены как подпрограммы в спецификации пакета (обычные процедуры и функции, а также функции реализующие действия знаков операций).

Все детали реализации внутреннего представления приватного типа доступны в теле пакета, и любая подпрограмма в теле пакета имеет к ним доступ и может модифицировать приватный тип как обычный тип. Таким образом, приватность типа сохраняется только вне пакета.

В данном примере необходимо обратить внимание на то, что спецификация пакета разделена на две части. Все что находится до зарезервированного слова **`private`** - это общедоступная часть описаний, которая будет "видна" всем пользователям пакета. Все что находится после зарезервированного слова **`private`** - это приватная часть описаний, которая будет "видна" только внутри пакета (и в его дочерних модулях; см. "Дочерние модули").

Может показаться противоречивым размещение приватных описаний в спецификации пакета. Действительно, мы пытаемся скрыть детали реализации приватного объекта, и размещаем их в спецификации пакета, которая доступна. Однако, это необходимо для программ, которые размещают экземпляры объектов приватного типа поскольку компилятор, благодаря такой информации, знает сколько необходимо зарезервировать места для размещения экземпляра объекта приватного типа.

Хотя читатель спецификации пакета видит как устроено реальное внутреннее представление реализации приватного типа, это не обеспечивает его возможностью явным и допустимым способом использовать эту информацию. При этом, примечательным является то, что экземпляры объектов приватного типа могут быть созданы вне пакета. Например:

```
with Accounts;          use Accounts;

procedure Demo_Accounts is

    Home_Account : Account;
    Mortgage     : Account;
    This_Account : Account;

begin
    Mortgage := Accounts.Create(Initial_Balance => 500.00);
    Withdraw(Home_Account, 50);

        . . .

    This_Account := Mortgage;       -- присваивание приватного типа - разрешено

    -- сравнение приватных типов
    if This_Account = Home_Account then

        . . .

end Demo_Accounts;
```

7.2.2 Лимитированные приватные типы (*limited private types*)

Хотя приватные типы позволяют разработчику получить значительный контроль над действиями

пользователя, ограничив способности пользователя в манипулировании объектами, бывают случаи когда необходимо запретить пользователю приватного типа использовать даже такие предопределенные операции как сравнение и присваивание.

В качестве демонстрации сказанного, рассмотрим следующий пример:

```
package Compare_Demo is

   type Our_Text is private;

   . . .

private
   type Our_Text (Maximum_Length : Positive := 20) is
      record
         Length : Index := 0;
         Value  : String(1..Maximum_Length);
      end record;

   . . .

end Compare_Demo;
```

Здесь, тип `Our_Text` описан как приватный и представляет из себя запись. В данной записи, поле длины `Length` определяет число символов которое содержит поле `Value` (другими словами - число символов которые имеют смысл). Любые символы, находящиеся в позиции от `Length + 1` до `Maximum_Length` будут нами игнорироваться при использовании этой записи. Однако, если мы попросим компьютер сравнить две записи этого типа, то он, в отличие от нас, не знает предназначения поля `Length`. В результате, он будет последовательно сравнивать значения поля `Length` и значения всех остальных полей записи. Очевидно, что алгоритм предопределенной операции сравнения в данной ситуации не приемлем, и нам необходимо написать собственную функцию сравнения.

Для подобных случаев Ада предусматривает лимитированные приватные типы. Изменим рассмотренный выше пример следующим образом:

```
package Compare_Demo is

   type Our_Text is limited private;

   . . .

   function "=" (Left, Right : in Our_Text) return Boolean;

   . . .

private
   type Our_Text (Maximum_Length : Positive := 20) is
      record
         Length : Index := 0;
         Value  : String(1..Maximum_Length);
      end record;
```

```
    ...
end Compare_Demo;
```

Теперь, тип `Our_Text` описан как лимитированный приватный тип. Также, в спецификации пакета, описана функция знака операции сравнения на равенство `"="`. Реализация алгоритма этой функции должна быть помещена в тело пакета. Примечательно, что при совмещении знака операции равенства `"="` автоматически производится неявное совмещение знака операции неравенства `"/="`. При этом следует учесть, что если функция реализующая действие знака операции равенства `"="` возвращает значение тип которого отличается от предопределенного логического типа `Boolean` (полное имя - `Standard.Boolean`), то совмещение знака операции неравенства `"/="` необходимо описать явно. Следует заметить, что Ада разрешает переопределять знак операции равенства для всех типов.

Для лимитированного приватного типа можно также создать процедуру для выполнения присваивания (или инициализации). Например, для показанного выше типа `Our_Text`, спецификация такой процедуры может иметь следующий вид:

```
    ...
    procedure Init (T : in out Our_Text;
                            S : in     String);
        . . .
```

Напомним, что спецификация такой процедуры должна быть размещена в спецификации пакета *Compare_Demo*, а ее тело (реализация) - в теле этого пакета.

7.2.3 Отложенные константы (*deferred constants*)

В некоторых спецификациях пакетов возникает необходимость описать константу приватного типа. Это можно выполнить таким же образом как и описание приватного типа (а также большинство опережающих ссылок). В общедоступной части спецификации пакета мы создаем неполное описание константы, после чего, компилятор ожидает получить полное описание константы в приватной части спецификации пакета. Например:

```
package Coords is


    type Coord is private;
    Home: constant Coord;  -- отложенная константа!

private
    type Coord is record
        X : Integer;
        Y : Integer;
    end record;


    Home : constant Coord := (0, 0);
end Coords;
```

В результате такого описания, пользователи пакета *Coords* "видят" константу `Home`, которая имеет приватный тип `Coord`, и могут использовать эту константу в своем коде. При этом,

детали внутреннего представления этой константы им не доступны и они могут о них не заботиться.

7.3 Дочерние модули (*child units*) (Ada95)

Не редко, во время интенсивной разработки большой системы, возникают случаи когда один из пакетов разрастается необычайно быстро, а его спецификация подвергается постоянным изменениям. Таким образом, ввиду частого обновления спецификации пакета, резко возрастают затраты на перекомпиляцию зависимых модулей, а увеличение размеров самого пакета усложняет его дальнейшую разработку и последующее сопровождение.

Для преодоления подобных трудностей Ада предлагает концепцию дочерних модулей, которая является основой в построении иерархических библиотек. Такой подход позволяет разделить пакет большого размера на самостоятельные пакеты и подпрограммы меньшего размера, объединенные в общую иерархию. Кроме того, эта концепция позволяет расширять уже существующие пакеты. Она предлагает удобный инструмент для обеспечения множества реализаций одного абстрактного типа и дает возможность разработки самодостаточных подсистем при использовании приватных дочерних модулей.

Дочерние модули непосредственно используются в стандартной библиотеке Ады. В частности, дочерними модулями являются такие пакеты как *Ada.Text_IO*, *Ada.Integer_Text_IO*. Следует также заметить, что концепция дочерних модулей была введена стандартом Ada95. В стандарте Ada83 эта идея отсутствует.

7.3.1 Расширение существующего пакета

Рассмотрим случай когда возникает необходимость расширения пакета который уже содержит некоторое множество описаний. Например, в пакете *Stacks* может понадобиться дополнительный сервис просмотра `Peek`. Если в текущий момент времени пакет *Stacks* используется многими модулями, то такая модификация, путем добавления нового сервиса, потребует значительных затрат на перекомпиляцию всех зависимых модулей, причем, включая и те модули которые не будут использовать новый сервис.

Следовательно, для логического расширения уже существующего пакета предпочтительнее использовать дочерний пакет который будет существовать абсолютно отдельно. Например:

```
package Stacks is

    type Stack is private;
    procedure Push(Onto : in out Stack; Item : Integer);
    procedure Pop(From : in out Stack; Item : out Integer);
    function Full(Item : Stack) return Boolean;
    function Empty(Item : Stack) return Boolean;

private
    -- скрытая реализация стека
    ...
    -- точка A

end Stacks;

package Stacks.More_Stuff is
```

```
        function Peek(Item : Stack) return Integer;

    end Stacks.More_Stuff;
```

По правилам Ады, спецификация родительского пакета обеспечивает для дочернего пакета такую же самую область видимости, что и для своего тела. Следовательно, дочерний пакет видит всю приватную часть спецификации родительского пакета.

В показанном выше примере, пакет *Stacks.More_Stuff* является дочерним пакетом для пакета *Stacks*. Значит, дочерний пакет *Stacks.More_Stuff* "видит" все описания пакета *Stacks*, вплоть до точки *A*.

Необходимо заметить, что для сохранения приватности описаний родительского пакета, Ада не позволяет включать в спецификацию дочернего пакета приватную часть спецификации родительского пакета.

Примечание:
Согласно правил именования файлов, принятым в системе компилятора *GNAT*, спецификация и тело пакета *Stacks* должны быть помещены в файлы:

`stacks.ads` и `stacks.adb`

соответственно, а спецификация и тело дочернего пакета *Stacks.More_Stuff* - в файлы:

`stacks-more_stuff.ads` и `stacks-more_stuff.adb`

Клиенту, которому необходимо использовать функцию `Peek`, просто необходимо включить дочерний пакет в инструкцию спецификатора совместности контекста **`with`**:

```
    with Stacks.More_Stuff;

    procedure Demo is


        X : Stacks.Stack;

    begin
        Stacks.Push(X, 5);
        if Stacks.More_Stuff.Peek = 5 then
            . . .

    end Demo;
```

Следует заметить, что включение дочернего пакета в инструкцию спецификатора совместности контекста **`with`** автоматически подразумевает включение в инструкцию **`with`** всех пакетов-родителей. Однако, инструкция спецификатора использования контекста **`use`** таким образом не работает. То есть, область видимости может быть получена пакетом только в базисе пакета.

```
    with Stacks.More_Stuff; use Stacks; use More_Stuff;

    procedure Demo is


        X : Stack;

    begin
        Push(X, 5);
        if Peek(x) = 5 then
            . . .
```

```
end Demo;
```

Необходимо также заметить, что подпрограммы (процедуры и функции) могут быть дочерними модулями пакета (правила их использования достаточно очевидны). При этом, однако, сами подпрограммы не могут иметь дочерние модули.

7.3.2 Иерархия модулей как подсистема

Каждый программный модуль (процедура, функция пакет) должен иметь уникальное имя. Бывают случаи, когда происходит быстрое заполнение пространства имен. Например, при проектировании большой системы достаточно сложно обеспечить уникальность имен и при этом сохранить их смысловое значение, а в случаях когда разные части проекта разрабатываются разными программистами (или даже коллективами программистов) риск получения коллизии имен увеличивается еще больше.

В подобных ситуациях, используя концепцию дочерних модулей Ады, можно выделить каждую отдельно разрабатываемую подсистему в самостоятельную иерархию модулей. При этом, каждая подсистема будет иметь свой собственный корневой пакет с уникальным именем.

```
package Root is

   -- корневой пакет может быть пустым

end Root;
```

Поскольку имя корневого пакета уникально, то имена дочерних модулей иерархии также будут уникальны, что, в результате, минимизирует вероятность коллизии имен. Кроме того, такой прием является удобным средством разделения большого проекта на логически самостоятельные составные части.

Примером использования подобного подхода может служить стандартная библиотека Ады, которая предстявляется как набор дочерних модулей трех корневых пакетов: *Ada*, *Interfaces* и *System*.

7.3.3 Приватные дочерние модули (*private child units*)

Дочерние модули могут быть приватными. Такая концепция позволяет создавать дочерние модули, которые будут видимы только внутри иерархии родительского пакета. В этом случае сервисы для подсистемы могут быть инкапсулированы внутри приватных пакетов, используя наследуемые ими преимущества для компиляции и видимости.

В спецификацию приватных дочерних пакетов разрешается включать приватную часть спецификации их пакетов-родителей, поскольку такие дочерние модули - приватны. В обычном случае - это не разрешается, так как нарушает сокрытие деталей реализации родительской приватной части.

```
private package Stacks.Statistics is

   procedure Increment_Push_Count;

end Stacks.Statistics;
```

Процедура *Stacks.Statistics.Increment_Push_Count* могла бы быть вызвана внутри реализации пакета *Stacks*. Такая процедура не будет доступна ни одному внешнему, по отношению к этой иерархии модулей, клиенту.

8. Переименования

Ада предоставляет программисту возможность осуществлять переименования. Следует заметить, что переименование иногда вызывает споры в организациях программирующих на Аде. Некоторым людям переименование нравится, а другим - нет. Существует несколько важных вещей, которые необходимо понять:

- Переименование не создает нового пространства для данных. Оно просто создает новое имя для уже присутствующей сущности.
- Не следует постоянно переименовывать одно и то же. Этип можно запутать всех, включая самого себя.
- Переименование необходимо использовать для упрощения кода. Введение нового имени, в некоторых случаях, делает код более легко читаемым.

При использовании переименований следует учитывать, что частое переименование объектов и их значений может создать трудности в понимании исходного текста. Хотя каждое новое имя может иметь определенный смысл в контексте нового пакета, при большом количестве последующих переименований становиться трудно отследить имя оригинала.

8.1 Уменьшение длин имен

Переименование может быть полезно в случае наличия длинных имен пакетов:

```
with  Ada.Text_IO;
with  Ada.Integer_Text_IO;

procedure Gun_Aydin is

   package TIO renames Ada.Text_IO;
   package IIO renames Ada.Integer_Text_IO;

      . . .
```

В этом случае, для внутреннего использования, длинное имя `Ada.Text_IO` переименовано в короткое имя `TIO`, а длинное имя `Ada.Integer_Text_IO` переименовано в короткое имя `IIO`.

8.2 Переименование знаков операций

В некоторых случаях, знак операции для типа описанного в пакете, который указан в спецификаторе контекста **`with`**, не является непосредственно видимым. В действительности, правила Ады заключаются в том, что сущность, находящаяся в контексте, не будет непосредственно видимой до тех пор, пока не будет явно указано, что она непосредственно видима. Спецификатор использования **`use`** для пакета всегда делает непосредственно видимыми знаки операций операции для типа описанного в пакете, однако, спецификатор использования **`use`** одновременно делает непосредственно видимыми все публично доступные ресурсы пакета, что может оказаться не желательным.

Переименование позволяет явно импортировать только те знаки операций, которые реально необходимы. При этом, видимость всех остальных ресурсов пакета остается не тронутой. Следующий пример показывает как это можно выполнить:

```
with  Ada.Text_IO;

procedure diamond1 is

    package TIO renames  Ada.Text_IO;

    function "+" (L, R: TIO.Count) return TIO.Count renames TIO."+";
    function "-" (L, R: TIO.Count) return TIO.Count renames TIO."-";

        . . .
```

Использование знаков операций облегчается при предварительном планировании использования знаков операций в процессе разработки пакета. В следующем примере знаки операций переименовываются во вложенном пакете, который, в последствии, может быть сделан непосредственно выдимым с помощью спецификатора использования **use**:

```
package Nested  is

    type T1     is private;
    type Status is (Off, Low, Medium, Hight);

    package Operators is

        function ">=" (L, R: Status) return Boolean renames Nested.">=";
        function "="  (L, R: Status) return Boolean renames Nested."=";

    end Operators;

private
    type T1 is ...
    ...
end Nested;
```

Показанный выше, вложенный пакет может быть сделан доступным путем указания спецификатора контекста "**with** Nested;", и последующего спецификатора использования "**use** Nested.Operators;" следующим образом:

```
with Nested;

procedure Test_Nested is

    use Nested.Operators;
    ...
begin
      . . .
```

Возможно, что не все одобрительно встретят подобную технику, однако она упрощает использование инфиксной формы знаков операций, поскольку позволяет отказаться от локального переименования. Следует заметить, что такое решение будет лучше чем

использование спецификатора использования типа "**use type**", поскольку делает видимым только ограниченное множество знаков операций. Однако, такой подход требует дополнительных усилий от разработчика пакета.

8.3 Переименование исключений

В некоторых случаях полезно осуществить локальное переименование исключения:

```
with  Ada.IO_Exceptions;

package My_IO is


    Data_Error: exception renames Ada.IO_Exceptions.Data_Error;

    ...
end My_IO;
```

8.4 Переименование компонентов

Наиболее часто забываемым свойством переименования Ады является возможность предоставления собственного имени определенному компоненту составного типа:

```
with  Ada.Text_IO;

package Rename_A_Variable is


    Record_Count: renames Ada.Text_IO.Count;

    ...
end Rename_A_Variable;
```

8.4.1 Переименование отрезка массива

Предположим, что у нас есть следующая строка:

```
Name  : String(1..60);
```

Причем, отрезок `(1..30)` - это фамилия (*last name*), отрезок `(31..59)` - имя (*first name*), символ в позиции `60` - это инициал отчества (*middle name*). Используя переименования мы можем выполнить следующее:

```
declare
    Last   : String renames Name(1..30);
    First  : String renames Name(31..59);
    Middle : String renames Name(60..60);
begin
    Ada.Text_IO.Put_Line(Last);
    Ada.Text_IO.Put_Line(First);
    Ada.Text_IO.Put_Line(Middle);
end;
```

В результате, каждый вызов `Put_Line` будет обращаться к именованному объекту, а не к диапазону индексов. При этом не осуществляется распределение дополнительного пространства для данных, а обеспечивается новое имя для доступа к уже существующим данным.

8.4.2 Переименование поля записи

Предположим, что у нас имеются следующие описания:

```
subtype Number_Symbol     is Character range '0' .. '9';
subtype Address_Character is Character range
   Ada.Characters.Latin_1.Space .. Ada.Characters.Latin_1.LC_Z;

type  Address_Data  is array (Positive range <>) of Address_Character;
type  Number_Data   is array (Positive range <>) of Number_Symbol;

type  Phone_Number  is
   record
      Country_Code  : Number_Data(1..2);
      Area_Code     : Number_Data(1..3);
      Prefix        : Number_Data(1..3);
      Last_Four     : Number_Data(1..4);
   end record;

type  Address_Record  is
   record
      The_Phone        : Phone_Number;
      Street_Address_1 : Address_Data(1..30);
      Street_Address_2 : Address_Data(1..20);
      City             : Address_Data(1..25);
      State            : Address_Data(1..2);
      Zip              : Number_Data(1..5);
      Plus_4           : Number_Data(1..4);
   end record;


One_Address_Record  : Address_Record;
```

Используя переименование, мы можем переименовать один из внутренних компонентов переменной записи `One_Address_Record` типа `Address_Record`, для прямого использования в программе. Например, мы можем переименовать `Area_Code` в инструкции блока:

```
declare
   AC: Number_Data renames One_Address_Record.The_Phone.Area_Code;
begin
   ...
end;
```

Описание `AC` не требует никакого распределения дополнительного пространства данных. Вместо этого, оно локализует имя для компонента, который вложен в запись. При наличии компонентов записей с большим уровнем вложения, такой подход может оказаться весьма удобным.

8.5 Переименование библиотечного модуля

Предположим, что в нашей библиотеке есть пакет который часто используется, и предположим, что этот пакет имеет довольно длинное имя. Пользуясь переименованием, мы можем указать этот пакет в спецификаторе контекста **`with`**, после чего, переименовать пакет с длинным

именем, и скомпилировать полученный модуль с более коротким именем обратно в библиотеку. Например:

```
with  Graphics.Common_Display_Types;
package CDT renames Graphics.Common_Display_Types;
```

Далее, мы можем использовать библиотечный модуль `CDT`, с более коротким именем, также как и библиотечный модуль `Graphics.Common_Display_Types`. При этом следует избегать переименований, когда новое имя очень сильно отличается от оригинала.

9. Настраиваемые модули в языке Ада (*generics*)

Долгие годы одной из самых больших надежд программистов была надежда в многократном повторном использовании однажды написанного кода (похоже, ...и осталась). Хотя библиотеки математических подпрограмм используются достаточно широко, следует заметить, что математика оказалась весьма удачной предметной областью в которой функции хорошо описаны и стабильны. Попытки разработать подобные библиотеки в других предметных областях имели достаточно ограниченный успех из-за неминуемого изменения алгоритмов обработки и типов данных в подпрограммах. Ада пытается помочь в решении этой проблемы предоставляя возможность производства кода, который в большей степени зависит не от специфики используемого типа данных, а от общности алгоритма обработки.

Как описано Найдичем (*Naiditch*), настраиваемые модули во многом подобны шаблонам стандартных писем. Шаблоны стандартных писем, в основном, являются законченными письмами, но с несколькими пустыми местами в которых необходимо произвести подстановку (например, дописать имя и адрес). Таким образом, шаблоны стандартных писем не могут быть отосланы до тех по пока не будут заполнены пустые места, которые должны содержать эту недостающую информацию.

Подобным образом, настраиваемые модули не могут быть непосредственно использованы. Мы создаем новую подпрограмму или пакет путем конкретизации (*instantiating*) настраиваемого модуля, или, другими словами, создаем из настраиваемого модуля экземпляр настроенного модуля. При конкретизации настраиваемого модуля мы должны предоставить недостающую информацию, которая может быть информацией о типе, значении или даже подпрограмме.

9.1 Общие сведения о настраиваемых модулях

В языке Ада, любой программный модуль (подпрограмма или пакет) может быть настраиваемым модулем. Такой настраиваемый модуль используется для создания экземпляра кода, который будет работать с фактическим типом данных. Требуемый тип данных передается как параметр настройки при конкретизации настраиваемого модуля (создании экземпляра настроенного модуля). Как правило, описание настраиваемого модуля представлено двумя частями: спецификацией настраиваемого модуля и телом настраиваемого модуля. Спецификация описывает интерфейс настраиваемого модуля, а тело содержит детали его внутренней реализации.

Однажды скомпилированные настраиваемые модули помещаются в библиотеку Ады и могут быть указаны в инструкции спецификатора совместности контекста **`with`** в других компилируемых модулях. При этом, следует заметить, что настраиваемые модули не могут быть указаны в инструкции спецификатора использования контекста **`use`**.

После указания настраиваемого модуля в спецификаторе контекста **`with`**, программный модуль

(подпрограмма, пакет или другой настраиваемый модуль) осуществляет конкретизацию настраиваемого модуля, то есть, создает экземпляр настроенного модуля из настраиваемого модуля. После этого, экземпляр настроенного модуля (конкретизированная подпрограмма или пакет) может быть сохранен в библиотеке Ады для последующего использования.

В качестве простого примера использования настраиваемого модуля, рассмотрим конкретизацию стандартного настраиваемого пакета *Integer_IO*:

```
with Ada.Text_IO;        use Ada.Text_IO;

package Int_IO is new Integer_IO(Integer);
```

Получившийся экземпляр настроенного модуля (пакет *Int_IO*), в последствии, может быть помещен в инструкции спецификации контекста **`with`** и **`use`** любого программного модуля.

Следует заметить, что стандартный настраиваемый пакет *Integer_IO* таким же образом может быть конкретизирован при использовании других типов.

```
with Ada.Text_IO;    use Ada.Text_IO;
with Accounts;       use Accounts;

package Account_No_IO is new Integer_IO(Account_No);
```

9.1.1 Настраиваемые подпрограммы

Рассмотрим простой пример описания настраиваемой подпрограммы. Однажды скомпилированная, она в последствии может быть помещена в инструкцию спецификатора совместности контекста **`with`** в других программных модулях. Напомним также, что настраиваемый модуль нельзя указывать в инструкции спецификатора использования контекста **`use`**.

Спецификация модуля настраиваемой подпрограммы содержит ключевое слово **`generic`**, после которого следует список формальных параметров настройки и далее - спецификация процедуры:

```
generic
    type Element is private;  -- Element - это параметр настраиваемой
                              -- подпрограммы

procedure Exchange(A, B : in out Element);
```

Тело настраиваемой процедуры описывает реализацию алгоритма работы процедуры и представляется как отдельно компилируемый модуль. Примечательно, что оно идентично не настраиваемой версии.

```
procedure Exchange(A, B  : in out Element) is


    Temp : Element;


begin
    T := Temp;
    T := B;
    B := Temp;
end Exchange;
```

Код, показанный выше, является простым шаблоном для процедуры, которая может быть

реально создана. Следует обратить внимание на то, что этот код не может быть непосредственно вызван. Это подобно описанию типа, поскольку не производится никакого распределения пространства памяти или генерации машинного кода.

Для создания реальной процедуры, то есть, экземпляра процедуры которую можно вызвать, необходимо выполнить конкретизацию настраиваемой процедуры:

```
procedure Swap is new Exchange(Integer);
```

Следует заметить, что в показанном выше примере конкретизация настраиваемой процедуры осуществлена с использованием простого позиционного сопоставления формального и фактического параметров настройки. В дополнение к позиционному сопоставлению, Ада позволяет использовать именное сопоставление формальных и фактических параметров настройки (при большом количестве параметров настройки, именное сопоставление улучшает читабельность). Показанную выше конкретизацию настраиваемой процедуры можно осуществить с помощью использования именного сопоставления следующим образом:

```
procedure Swap is new Exchange(Element => Integer);
```

Теперь мы имеем процедуру `Swap` которая меняет местами переменные целого типа `Integer`. Здесь, `Integer` является фактическим параметром настройки, а `Element` - формальным параметром настройки.

Процедура `Swap` может быть вызвана (и она будет вести себя) как будто она была описана следующим образом:

```
procedure Swap (A, B  : in out Integer) is

   Temp : Integer;

begin

   ...

end Swap;
```

Таких процедур `Swap` можно создать столько, сколько необходимо.

```
procedure Swap is new Exchange(Character);
procedure Swap is new Exchange(Element => Account); -- ассоциация по имени
```

В этом случае будет нормально использоваться совмещение имен процедур. Компилятор будет определять к какой конкретно процедуре производится вызов используя информацию о типе параметра.

9.1.2 Настраиваемые пакеты

Пакеты также могут быть настраиваемыми. Следующая спецификация настраиваемого пакета достаточно традиционна:

```
generic
   type Element is private; -- примечательно, что это параметр
                            -- настройки
package Stacks is
   procedure Push(E : in Element);
```

```
    procedure Pop(E : out Element);
    function Empty return Boolean;
private
    The_Stack : array(1..200) of Element;
    top       : Integer range 0..200 := 0;
end Stacks;
```

Сопутствующее тело пакета может иметь подобный вид:

```
package body Stacks is

    procedure Push(E : in Element) is
        ...

    procedure Pop(E : out Element) is
        ...

    function Empty return Boolean is
        ...

end Stacks;
```

В качестве элемента настройки, необходимо просто указать любой экземпляр типа данных.

9.1.3 Дочерние настраиваемые модули

Настраиваемые пакеты, подобно обычным пакетам Ады, могут иметь дочерние модули. При этом, следует заметить, что такие дочерние модули также должны быть настраиваемыми модулями.

В качестве примера, предположим, что нам необходимо расширить настраиваемый пакет *Stacks*, который был показан в примере выше (см. 9.1.2). Допустим, что нам необходимо добавить функцию `Top`, которая возвращает объект находящийся в вершине стека, но при этом не удаляет его из стека. Чтобы решить эту задачу, мы можем, для настраиваемого пакета *Stacks*, описать дочерний настраиваемый пакет *Stacks.Additions*. Спецификация *Stacks.Additions* может выглядеть следующим образом:

```
generic
package Stacks.Additions is
    function Top return Element;
end Stacks.Additions;
```

Примечательно, что дочерний настраиваемый модуль "видит" все компоненты своего родителя, включая все параметры настройки.

Тело дочернего настраиваемого модуля *Stacks.Additions* может иметь следующий вид:

```
package body Stacks.Additions is

    function Top return Element is
        ...

end Stacks.Additions;
```

Ниже демонстрируется пример конкретизации настраиваемых модулей *Stacks* и *Stacks.Additions*.

Конкретизация модуля *Stacks* формирует пакет *Our_Stacks*, что имеет вид:

```
with  Stacks;

package Our_Stack is new Stack(Integer);
```

Конкретизация модуля *Stacks.Additions* формирует пакет *Our_Stack_Additions*, что имеет вид:

```
with  Our_Stack, Stacks.Additions;

package Our_Stack_Additions is new Stacks.Additions;
```

Примечательно, что настраиваемый дочерний модуль рассматривается как описанный внутри настраиваемого родителя.

9.2 Параметры настройки для настраиваемых модулей

Существует три типа параметров для настраиваемых модулей:

- параметры-типы
- параметры-значения
- параметры-подпрограммы

Необходимо заметить, что до настоящего момента в примерах мы рассматривали только параметры-типы.

9.2.1 Параметры-типы

Не смотря на привлекательные свойства настраиваемых модулей, при определении характера задач, решаемых с помощью настраиваемого модуля, необходимо иметь возможность накладывать некоторые ограничения. Допустим, что некоторые настраиваемые модули предусматривают возможность суммирования массива чисел. Очевидно, что это характерно только для чисел, и мы не можем производить суммирование записей. Следовательно, для того, чтобы защититься от конкретизации настраиваемых модулей с указанием не подходящих типов данных, требуется возможность в установке некоторых ограничений. Кроме того, желательно чтобы компилятор также мог осуществлять проверку того, что мы не выполняем какие-нибудь не допустимые действия с переменной внутри кода настраиваемого модуля, например, чтобы он не разрешал использовать атрибут **`'Pred`** для записей.

Решение таких задач обеспечивается механизмом который основан на виде спецификации формального параметра-типа. Таким образом, спецификация формального параметра-типа определяет категорию типов, которые могут быть использованы при конкретизации настраиваемого модуля, а также те действия, которые можно осуществлять над формальным параметром внутри настраиваемого модуля. Ниже показан общий список вариантов спецификаций формальных параметров-типов и различные ограничения, накладываемые на выбор фактических параметров-типов настраиваемых модулей.

```
type T is limited private     -- тип T - любой тип
type T is private             -- тип T - любой не лимитированный тип



type T is (<>)                -- тип T любой дискретный тип
                      -- (целочисленный или перечислимый)
type T is range <>            -- тип T любой целочисленный тип
type T is mod <>              -- тип T любой модульный целочисленный тип
```

```
type T is digits <>           -- тип T любой вещественный тип с плавающей точкой
type T is delta <>            -- тип T любой вещественный тип с фиксированной точкой
type T is delta <> digits <> -- тип T любой вещественный децимальный тип



type T is access Y;           -- тип T любой ссылающийся на Y ссылочный тип
type T is access all Y;       -- тип T любой "access all Y" ссылочный тип
type T is access constant Y; -- тип T любой "access constant Y" ссылочный тип

         -- примечание: тип Y может быть предварительно описанным

         --       настраиваемым параметром



type T is array(Y range <>) of Z; -- тип T любой неограниченный массив элементов типа Z

                          -- у которого Y - подтип индекса
type T is array(Y) of Z;           -- тип T любой ограниченный массив элементов типа Z

                          -- у которого Y - подтип индекса

         -- примечание: тип Z (тип компонента фактического массива)

         --       должен совпадать с типом формального массива.

         --       если они не являются скалярными типами,

         --       то они оба должны иметь тип

         --       ограниченного или неограниченного массива



type T is new Y;                          -- тип T любой производный от Y тип
type T is new Y with private;             -- тип T любой не абстрактный тэговый тип

                                  -- производный от Y
type T is abstract new Y with private;    -- тип T любой тэговый тип производный от Y
type T is tagged private;                 -- тип T любой не абстрактный не лимитированый

                                  -- тэговый тип
type T is tagged limited private;         -- тип T любой не абстрактный тэговый тип
type T is abstract tagged private;        -- тип T любой не лимитированый тэговый тип
type T is abstract tagged limited private; -- тип T любой тэговый тип
```

Для того чтобы лучше понять правила управляющие конкретизацией для параметров типа массив, рассмотрим следующий настраиваемый пакет:

```
generic
   type Item  is private;
   type Index  is (<>);
   type Vector is array (Index range <>) of Item;
   type Table  is array (Index) of Item;
package P is . . .
```

и типы:

```
type Color is (red, green, blue);
type Mix is array (Color range <> ) of Boolean;
type Option is array (Color) of Boolean;
```

тогда, `Mix` может соответствовать `Vector`, а `Option` может соответствовать `Table`.

```
package R is new P( Item   => Boolean,
                    Index  => Color,
                    Vector => Mix,
                              Table  => Option);
```

9.2.2 Параметры-значения

Параметры-значения позволяют указывать значения для переменных внутри настраиваемого модуля:

```
generic

    type Element is private;
    Size: Positive := 200;

package Stacks is

        procedure Push...
        procedure Pop...
        function Empty return Boolean;

end Stacks;

package body Stacks is

    Size : Integer;
    theStack : array(1..Size) of Element;

            . . .
```

Тогда, создать экземпляр настраиваемого модуля можно одним из следующих способов:

```
package Fred is new Stacks(Element => Integer, Size => 50);

package Fred is new Stacks(Integer, 1000);

package Fred is new Stacks(Integer);
```

Следует обратить внимание на то, что при конкретизации настраиваемого модуля фактический параметр-значение должен быть обязательно указан только в случаях когда для формального параметра-значения не представлено значение по-умолчанию.

В качестве параметров-значений допускается использование строк.

```
generic

    type Element is private;
    File_Name : String;

package ....
```

Примечательно, что параметр `File_Name`, имеющий строковый тип `String`, - не ограничен (*not constrained*). Это идентично строковым параметрам для подпрограмм.

9.2.3 Параметры-подпрограммы

В качестве параметра настройки для настраиваемого модуля может быть передана подпрограмма. Необходимость в параметрах-подпрограммах чаще всего возникает когда какой-либо формальный параметр-тип настраиваемого модуля описан как приватный или лимитированный приватный тип. В таких случаях, Ада накладывает традиционные ограничения

на использование операций над экземплярами данного типа внутри тела настраиваемого модуля. Однако, при этом может возникнуть необходимость в осуществлении сравнения или выполнения проверки на равенство значений данного типа, внутри тела настраиваемого модуля. Следовательно, параметры-подпрограммы являются механизмом, который предоставляет для компилятора информацию о том как осуществлять эти действия.

В качестве примера рассмотрим типичный случай требующий применение параметров-подпрограмм. Предположим, что в качестве одного из параметров настраиваемого модуля используется лимитированый приватный тип. Тогда, для этого лимитированного приватного типа, с помощью параметров-подпрограмм, можно осуществить передачу операций: проверка на равенство и присваивание.

```
generic


   type Element is limited private;
   with function "="(E1, E2 : Element) return Boolean;
   with procedure Assign(E1, E2 : Element);

package Stuff is . . .
```

Конкретизация такого настраиваемого модуля может иметь вид:

```
package Things is new Stuff(Person, Text."=", Text.Assign);
```

Для формального параметра-подпрограммы может быть указана подпрограмма используемая по-умолчанию. Тогда, при конкретизации настраиваемого модуля, фактический параметр-подпрограмма может не указываться. Предположим, что у нас есть подпрограмма, спецификация которой имеет вид:

```
procedure My_Assign(E1, E2 : Person);
```

Тогда, при описании формального параметра-подпрограммы `Assign`, мы можем указать процедуру `My_Assign` как подпрограмму которую необходимо использовать по-умолчанию следующим образом:

```
generic


   type Element is limited private;
   with function "="(E1, E2 : Element) return Boolean;
   with procedure Assign(E1, E2 : Element) is My_Assign(E1, E2 : Person);

package Stuff is . . .
```

В результате, конкретизация настраиваемого модуля, с использованием подпрограммы заданной по-умолчанию, может иметь следующий вид:

```
package Things is new Stuff(Person, Text."=");
```

И наконец, при описании спецификации формального параметра-подпрограммы, можно указать, что выбор процедуры по-умолчанию должен осуществляться согласно традиционных правил выбора подпрограмм. Для функции, реализующей действие знака проверки на равенство (`"="`) мы можем указать это следующим образом:

```
generic
```

```
type Element is limited private;
with function "="(E1, E2 : Element ) return Boolean is <>;
   . . .
```

Теперь, если при конкретизации настраиваемого модуля для функции `"="` не будет представлена соответствующая функция, то будет использоваться функция проверки на равенство по-умолчанию, выбранная в соответствии с фактическим типом `Element`. Например, если фактический тип `Element` - тип `Integer`, то будет использоваться обычная, для типа `Integer`, функция `"="`.

9.3 Преимущества и недостатки настраиваемых модулей

В заключение обсуждения настраиваемых модулей Ады необходимо отметить преимущества и недостатки использования данной концепции.

Основным преимуществом использования настраиваемых модулей является то, что они оказывают значительное содействие в многократном повторном использовании ранее разработанных и отлаженных алгоритмов. Действительно, настраиваемые модули позволяют разработчику однажды написать и отладить алгоритмы подпрограмм для обработки объектов тип которых в последствии будет указываться пользователями этих подпрограмм.

Однако, применение настраиваемых модулей не лишено недостатков. Разработка алгоритмов для настраиваемых модулей требует более тщательного внимания, что в результате является дополнительной нагрузкой для их автора. Также необходимо заметить, что реализация настраиваемой процедуры, функции или пакета может оказаться не столь эффективной как непосредственная реализация. Компилятор может генерировать один и тот же код для всех экземпляров настроенных процедур, функций или пакетов, не зависимо от фактически обрабатываемых данных.

10. Исключения

Как это не печально, но процесс разработки и эксплуатации любого программного обеспечения всегда сочетается с процессом поиска и исправления ошибок. Все ошибки, возникающие в программах на языке Ада, можно разделить на два класса:

- ошибки, которые обнаруживаются на этапе компиляции программы
- ошибки, которые обнаруживаются во время выполнения программы

Таким образом, не смотря на то, что одной из целей при разработке Ады была задача максимально обеспечить возможность ранней диагностики и обнаружения ошибок, то есть обнаружение ошибок на стадии компиляции программы, бывают случаи когда ошибки возникают и во время выполнения программы.

В Аде, ошибочные или другие исключительные ситуации, возникающие в процессе выполнения программы, называются исключениями. Это значит, что при возникновении ошибочной ситуации, во время выполнения программы, вырабатывается сигнал о наличии исключения. Такое действие называют возбуждением (генерацией или порождением) исключения и понимают как приостановку нормального выполнения программы для обработки соответствующей ошибочной ситуации. В свою очередь, обработка исключения - это выполнение соответствующего кода для определения причины возникновения ошибочной ситуации которая привела к возбуждению исключения, а также, при возможности, устранение

причины возникновения ошибки и/или выполнение других корректирующих действий.

Примечательно, что идея использования механизма исключений - это тема многих споров о том, что исключения являются или путем "ленивого" программирования, без достаточного анализа проблем и сопутствующих условий приводящих к возникновениям ошибок, или обычным видом управляющих структур, которые могут быть использованы для достижения некоторых эффектов. Тем не менее, хотя эти споры не прекращаются и в настоящее время, следует обратить внимание на то, что механизм исключений благополучно заимствован некоторыми современными реализациями других языков программирования (например, широко известная реализация языка *Object Pascal* фирмы *Borland*).

Все исключения языка программирования Ада можно разделить на стандартно предопределенные исключения и исключения определяемые пользователем.

10.1 Предопределенные исключения

Существует пять исключений которые стандартно предопределены в языке программирования Ада:

Constraint_Error	-	Ошибка ограничения
Numeric_Error	-	Ошибка числа
Program_Error	-	Ошибка программы
Storage_Error	-	Ошибка памяти
Tasking_Error	-	Ошибка задачи

10.1.1 Исключение *Constraint_Error*

Исключение *Constraint_Error* возбуждается в следующих случаях:

- при попытке нарушения ограничения диапазона, ограничения индекса или ограничения дискриминанта
- при попытке использования компонента записи, не существующего при текущем значении дискриминанта
- при попытке использования именуемого или индексируемого компонента, отрезка или атрибута объекта, обозначенных ссылочным значением, если этот объект не существует, поскольку ссылочное значение равно **`null`**

Рассмотрим пример:

```
procedure Constraint_Demo is


   X : Integer range 1..20;
   Y : Integer;


begin
   Put("enter a number ");
   Get(Y);
   X := Y;
   Put("thank you");
end Constraint_Demo;
```

Если пользователь вводит значение выходящее за диапазон значаний `1..20`, то нарушается ограничение диапазона значений для `X`, и происходит исключение *Constraint_Error*. Поскольку в

этом примере не предусмотрен код, который будет обрабатывать это исключение, то выполнение программы будет завершено, и окружение времени выполнения Ады (Ада-система) проинформирует пользователя о возникшей ошибке. При этом, строка

```
Put("thank you");
```

выполнена не будет. Таким образом, при возникновении исключения, остаток, выполняющегося в текущий момент блока, будет отброшен.

Рассмотрим пример в котором выполняется нарушение ограничения диапазона индексных значений для массива:

```
procedure Constraint_Demo2 is

   X : array (1..5) of Integer := (1, 2, 3, 4, 5);
   Y : Integer := 6;

begin
   X(Y) := 37;
end Constraint_Demo2;
```

Здесь, исключение *Constraint_Error* будет генерироваться когда мы будем пытаться обратиться к несуществующему индексу массива.

10.1.2 Исключение *Numeric_Error*

Исключение *Numeric_Error* возбуждается в случае когда предопределенная численная операция не может предоставить математически корректный результат Это может произойти при арифметическом переполнении, делении на нуль, а также не возможности обеспечить требуемую точность при выполнении операций с плавающей точкой. Следует заметить, что в Ada95 *Numeric_Error* переопределена таким образом, что является тем же самым, что и *Constraint_Error.*

```
procedure Numeric_Demo is

   X : Integer;
   Y : Integer;

begin
   X := Integer'Last;
   Y := X + X;          -- вызывает Numeric_Error
end Numeric_Demo;
```

10.1.3 Исключение *Program_Error*

Исключение *Program_Error* возбуждается в следующих случаях:

- при попытке вызова подпрограммы, активизации задачи или конкретизации настройки, если тело соответствующего модуля еще не обработано.
- если выполнение функции достигло завершающего **`end`** так и не встретив инструкцию возврата (**`return ...`**)
- при межзадачном взаимодействии во время выполнения инструкции отбора с ожиданием (**`select ...`**), когда все альтернативы закрыты и отсутствует раздел **`else`**

Кроме того, это исключение может возбуждаться в случае возникновения ошибки элаборации.

```
procedure Program_Demo is


   Z : Integer;


   function Y(X : Integer) return Integer is
   begin
      if X < 10 then
         return X;
      elsif  X < 20 then
         return X
      end if;
   end Y;              -- если мы попали в эту точку, то это значит,
                              -- что return не был выполнен

begin
   Z := Y(30);
end Program_Demo;
```

10.1.4 Исключение *Storage_Error*

Исключение *Storage_Error* возбуждается в следующих случаях:

- при попытке размещения динамического объекта обнаруживается, что нет достаточного пространства в динамической памяти (куче) которая выделена для задачи
- при исчерпании памяти выделенной для набора (коллекции) динамически размещаемых объектов
- при обращении к подпрограмме, когда израсходовано пространство стека

10.1.5 Исключение *Tasking_Error*

Исключение *Tasking_Error* возбуждается в случаях межзадачного взаимодействия. Оно может быть возбуждено при возникновении какого-либо исключения внутри задачи которая в текущий момент времени принимает участие в межзадачном взаимодействии или когда задача пытается организовать рандеву с абортированной задачей.

10.2 Исключения определяемые пользователем

Механизм исключений Ады был бы не столь полным если бы он позволял использовать только стандартно предопределенные исключения. Поэтому, в дополнение к стандартно предопределенным исключениям, Ада дает программисту возможность описывать свои собственные исключения и, в случае необходимости, выполнять их возбуждение.

10.2.1 Описание исключения пользователя

Описания пользовательских исключений должны размещаться в декларативной части кода, то есть там где обычно размещаются описания (например, в спецификации пакета). Форма описания исключений достаточно тривиальна и имеет следующий вид:

```
My_Very_Own_Exception : exception;
```

```
Another_Exception     : exception;
```

10.2.2 Возбуждение исключений

Указание возбуждения исключения достаточно простое. Для этого используется инструкция **`raise`**. Например:

```
raise Numeric_Error;
```

Сгенерированное таким образом исключение не будет ничем отличаться от "истинного" исключения *Numeric_Error*.

10.3 Обработка исключений

В настоящий момент мы уже знаем стандартно предопределенные исключения Ады, знаем как описывать исключения и знаем как их возбуждать (и свои, и предопределенные). Однако, весь код примеров, которые мы рассматривали, не выполнял никакой обработки исключений. Во всех рассмотренных случаях, после возникновения исключения, происходило простое завершение выполнения программы кодом библиотеки времени выполнения Ады. Таким образом, для того, чтобы извлечь некоторую пользу из механизма исключений, необходимо знать как анализировать ситуацию в случае возникновения исключения. Проще говоря - необходимо рассмотреть как и где писать обработчики исключений.

10.3.1 Обработчики исключений

Обработчик исключения может размещаться в конце тела подпрограммы, пакета или настраиваемого модуля, в конце тела задачи или входа, а также в конце инструкции блока или инструкции принятия (**`accept`**). Заметим, что обработчик исключения не является обязательной частью этих конструкций.

Рассмотрим следующий пример:

```
declare


    X : Integer range 1..20;

begin
    Put("please enter a number ");
    Get(X);
    Put("thank you");

exception
    when Constraint_Error =>
        Put("that number should be between 1 and 20");
    when others =>
        Put("some other error occurred");
end;
```

Здесь описаны два обработчика. В одном выполняется обработка только исключений ограничения (*Constraint_Error*). Второй обработчик выполняет обработку всех остальных исключений (**`others`**). Таким образом, если пользователь вводит число в диапазоне от 1 до 20, то ошибки не происходит и появляется сообщение "`thank you`". В противном случае, перед

завершением выполнения появляется сообщение обработчика исключения *Constraint_Error*: "`that number should be between 1 and 20`". В случае возникновения какого-либо другого исключения появится сообщение от второго обработчика: "`some other error occurred`".

Можно описать обработчик исключений так, чтобы он обрабатывал несколько указанных исключений. Для выполнения этого, исключения должны разделяться символом '|':

```
   . . .
exception
   . . .
   when Constraint_Error | Storage_Error =>
            . . .
```

Также следует заметить, что обработчик **`when others`** всегда должен быть последним в списке обработчиков исключений.

Если мы хотим чтобы пользователь продолжал ввод чисел до тех пор пока не пропадет ошибка ограничения, мы можем переписать предыдущий пример подобным образом:

```
loop
   declare
      . . .

   begin
      . . .

      Get(X);
      exit;

   exception
      when Constraint_Error =>
        Put("that number ...
   end;

   . . .      -- здесь будет продолжено выполнение
              -- после возникновения исключения
              -- и обработки его обработчиком
end loop;
```

Кроме того, этот пример показывает точку в которой будет продолжено выполнение инструкций после возникновения исключения и его обработки обработчиком.

10.3.2 Распространение исключений

Для того, чтобы точно знать в каком месте должен быть расположен соответствующий обработчик исключения, необходимо понимать как при возникновении исключения, во время работы программы, происходит поиск обработчика. Этот процесс поиска называется распространением исключений.

Если исключение не обрабатывается в подпрограмме в которой это исключение возникло, то оно распространяется в подпрограмму которая вызвала текущую подпрорамму (на уровень выше).

После чего, обработчик исключения ищется в вызвавшей подпрограмме. Если обработчик исключения не найден, то исключение распространяется дальше (еще на уровень выше). Это продолжается до тех пор пока не будет найден обработчик возникшего исключения или не будет достигнут уровень выполнения текущей задачи (наивысший уровень). Если в случае достижения уровня выполнения текущей задачи (то есть наивысшего уровня) обработчик исключения не будет найден, то текущая задача будет аварийно завершена (другими словами, абортирована). Если выполняется только одна задача, то библиотека времени выполнения Ады выполняет обработку возникшего исключения и аварийно завершает выполнение всей программы (другими словами, абортирует выполнение программы).

```
procedure Exception_Demo is


   ---------------------------------
   procedure Level_2 is
      -- здесь нет обработчика исключений
   begin
      raise Constraint_Error;
   end Level_2;


   ---------------------------------
   procedure Level_1 is
   begin
      Level_2;
   exception
      when Constraint_Error =>
         Put("exception caught in Level_1");
   end Level_1;

begin
   Level_1;

exception
   when Constraint_Error =>
      Put("exception caught in Exception_Demo");
end Exception_Demo;
```

После запуска этой программы будет выдано только сообщение "`exception caught in Level_1`". Следовательно, обработанное исключение не распространяется дальше.

Модифицируем процедуру `Level_1` поместив инструкцию **`raise`** в ее обработчик исключения. Наш предыдущий пример будет иметь следующий вид:

```
procedure Exception_Demo is


   ---------------------------------
   procedure Level_2 is
      -- здесь нет обработчика исключений
   begin
      raise Constraint_Error;
   end Level_2;


   ---------------------------------
```

```
    procedure Level_1 is
    begin
        Level_2;
    exception
        when Constraint_Error =>
            Put("exception caught in Level_1");
            raise;      -- регенерация текущего исключения;
                        -- дает возможность другим подпрограммам
                        -- произвести обработку возникшего
                        -- исключения
    end Level_1;

begin
    Level_1;

exception
    when Constraint_Error =>
        Put("exception caught in Exception_Demo");
end Exception_Demo;
```

Теперь, инструкция **raise**, помещенная в обработчик исключения, вызывает распространение исключения *Constraint_Error* на один уровень выше, то есть, к вызвавшей подпрограмме. Таким образом, исключение может быть получено и соответствующим образом обработано в каждой подпрограмме иерархии вызовов.

Инструкцию **raise** очень удобно использовать в секции **others** обработчика исключений:

```
    . . .

exception

    . . .

    when others =>
        raise;      -- регенерация текущего исключения;
                    -- дает возможность другим подпрограммам
                    -- произвести обработку возникшего
                    -- исключения
end;
```

В этом случае, соответствующее исключение будет продолжать генерироваться и распространяться до тех пор, пока не будет обработано надлежащим образом.

10.3.3 Проблемы с областью видимости при обработке исключений определяемых пользователем

Во всех предыдущих примерах, посвященных обработке исключений, были использованы стандартно определенные исключения Ады. Обработка исключений определяемых пользователем идентична обработке предопределенных исключений, однако, при этом могут возникать некоторые проблемы с областью видимости. Рассмотрим следующий пример:

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Demo is

    procedure Problem_In_Scope is
        Cant_Be_Seen : exception;
    begin
        raise Cant_Be_Seen;
    end Problem_In_Scope;

begin
    Problem_In_Scope;

exception
    when Cant_Be_Seen =>
        Put("just handled an_exception");
end Demo;
```

Этот пример не корректен. Проблема в том, что область видимости исключения `Cant_Be_Seen` ограничивается процедурой `Problem_In_Scope`, которая, собственно и является источником этого исключения. То есть, исключение `Cant_Be_Seen` не видимо и о нем ничего не известно за пределами процедуры `Problem_In_Scope`. Поэтому, это исключение не может быть точно обработано процедурой `Demo`.

Решить эту проблему можно использованием опции **`others`** в обработчике исключений внешней процедуры `Demo`:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is

    procedure Problem_In_Scope is
            Cant_Be_Seen : exception;
    begin
            raise Cant_Be_Seen;
    end Problem_In_Scope;

begin
    Problem_In_Scope;

exception
    when others =>
        Put("just handled some exception");
end Demo;
```

Другая проблема возникает тогда, когда в соответствии с правилами области видимости, исключение, описываемое в одной процедуре, перекрывает (или прячет) исключение, описываемое в другой процедуре:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is

    Fred : exception;          -- глобальное исключение
```

```
    ------------------------------------
    procedure P1 is
    begin
        raise Fred;
    end P1;

    ------------------------------------
    procedure P2 is

        Fred : exception;       -- локальное исключение

    begin
        P1;

    exception
        when Fred =>
            Put("wow, a Fred exception");
    end P2;
    ------------------------------------

begin
    P2;

exception
    when Fred =>
        Put("just handled a Fred exception");
end Demo;
```

Выводом такой процедуры будет "`just handled a Fred exception`". Исключение, обрабатываемое в процедуре `P2`, будет локально описанным исключением. Такое поведение подобно ситуации с областью видимости обычных переменных.

Для решения этой проблемы, процедуру `P2` можно переписать следующим образом:

```
-------------------------------------
procedure P2 is


    Fred : exception;

begin
    P1;

exception
    when Fred =>
        -- локальное исключение
        Put("wow, an_exception");


    when Demo.Fred =>
        -- "более глобальное" исключение
        Put("handeled Demo.Fred exception");
        raise;
end P2;
```

Теперь, обработчик исключения процедуры `P2` выдаст сообщение "`handeled Demo.Fred exception`" и, с помощью инструкции **`raise`**, осуществит передачу исключения `Demo.Fred` в обработчик исключения процедуры `Demo`, который, в свою очередь, выдаст сообщение "`just handled a Fred exception`".

10.3.4 Пакет *Ada.Exceptions*

Стандартный пакет *Ada.Exceptions* предоставляет некоторые дополнительные средства, которые могут быть использованы при обработке исключений.

Описанный в нем объект:

```
Event : Exception_Occurence;
```

и подпрограммы:

- функция `Exception_Name(Event)` - возвращает строку имени исключения, начиная от корневого библиотечного модуля
- функция `Exception_Information(Event)` - возвращает строку детальной информации о возникшем исключении
- функция `Exception_Message(Event)` - возвращает строку краткого объяснения исключения
- процедура `Reraise_Occurence(Event)` - выполняет повторное возбуждение исключения `Event`
- процедура `Reraise_Exception(e, "Msg")` - выполняет возбуждение исключения `e` с сообщением "`Msg`"

Могут быть весьма полезны при необходимости обработки неожиданных исключений. В таких случаях можно использовать код который подобен следующему:

```
   ...
exception
   ...
   when The_Event: others =>
      Put("Unexpected exeption is ";
      Put(Exeption_Name(The_Event));
            New_Line;
```

10.4 Подавление исключений

10.4.1 Принципы подавления исключений

Как правило, существует два источника которые выполняют возбуждение исключений при обнаружении некоторых ошибочных условий. Один из них - это механизмы аппаратной проверки, которые зависят от конкретно используемого оборудования. Второй источник для нас более интересен, поскольку этим источником является дополнительный машинный код, который генерирует компилятор.

Поскольку генерация дополнительного кода выполняется компилятором, а все компиляторы языка Ада должны соответствовать стандартным требованиям, то должно быть обеспечено стандартное средство управления вставкой подобных проверок в результирующий машинный код. Таким средством Ады является директива компилятора `Supress` (подавление проверок).

Эта директива может быть размещена в том месте, где не требуется производить проверки. Подавление проверок будет распространяться до конца текущего блока (при этом используются обычные правила области видимости).

Директива `Supress` имеет большое количество опций, позволяющих подавлять проверки различного вида на уровне типа, объекта или на функциональном уровне. Следует заметить, что многогранность директивы `Supress` сильно зависит от реализации конкретного компилятора, и различные реализации компиляторов свободны в предоставлении (или игнорировании) любых свойств этой директивы.

Исключение *Constraint_Error* имеет несколько подавляемых проверок:

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Idex_Check);
pragma Suppress (Length_Check);
pragma Suppress (Range_Check);
pragma Suppress (Division_Check);
pragma Suppress (Owerflow_Check);
```

Исключение *Program_Error* имеет только одну подавляемую проверку:

```
pragma Suppress (Elaboration_Check);
```

Исключение *Storage_Error* также имеет только одну подавляемую проверку:

```
pragma Suppress (Storage_Check);
```

10.4.2 Выполнение подавления исключений

Мы можем подавить проверку исключений для индивидуального объекта:

```
pragma Suppress (Idex_Check, on => table);
```

Подавление проверки исключений также может относиться к какому-то определенному типу:

```
type Employee_Id is new Integer;
pragma Suppress (Range_Check, Employee_Id);
```

Более полным примером использования директивы `Supress` может служить код показанный ниже. В этом случае область действия директивы распространяется до конца блока.

```
declare

    pragma Suppress(Range_Check);
    subtype Small_Integer is Integer range 1..10;

    A : Small_Integer;
    X : Integer := 50;

begin
    A := X;
```

```
        end;
```

Этот код не будет генерировать ошибок ограничения (*Constraint_Error*).