Параллельное программирование

Для того чтобы поддержать механизм параллельного выполнения программ предлагается 3 варианта – встроенный в язык механизм, автоматическое распараллеливание компилятором и явное использование библиотек для параллельного программирования. Эти три механизма отличаются уровнем абстракции, чем выше уровень абстракции тем легче программировать и тем меньший выигрыш в скорости работы программы мы можем получить в общем случае. Итак начнем с языкового механизма.

Ключевое слово concurrent

Определяя атрибут с характеристикой concurrent мы в явном виде указываем что все операции, которые будут выполняться при вызове подпрограмм или при обращении к атрибутам будут иметь несколько отличную семантику от обычных обращений. А именно, что выполнение любого обращение означает активация отличного от текущего контекста выполнения и само такое выполнение может приводить к ожиданию готовности атрибута к выполнению операции или постановку вызова процедуры в очередь на выполнение к данному атрибуту. Давайте рассмотрим примеры

Начнем с описаний

**concurrent unit** Process

procedure **is** … **end**

**end**

**unit** Actor

function: Type **is** … **end**

constant: Type **is** …

variable: Type

**end**

a: **concurrent** Actor **is** Actor

p **is** Process

Таким образом переменные a и p являются ссылками на параллелшьные объекты, которые существуют параллельно с текущим потоком управления. И с ними можно работать как с обычными переменными.

a.procedure

Это вроде обычный вызов процедуры от объекта, на который при выполнении будет указывать a, за исключение того что любой вызов процедуры из параллельного объекта есть асинхронный вызов. Т.е. a.procedure не приводит к ожиданию выполнения всего тела процедуры, а просто запускает ее на выполнение или более точно ставит запрос на выполнение процедуры procedure для объекта, с которым связан a.

А если мы вызываем функцию или обращаемся к константному или переменному атрибуту, то это точка синхронизации – мы дожидаемся пока объект, связанный с а будет готов выполнить запрос и дожидаемся конца выполнения запроса и забираем результат этого запроса в свою нить выполнения. По сходной схеме работают процедуры инициализации – объект не является готовым к выполнению обращений к своим подпрограммам и атрибутам до тех пор, пока процедура инициализации не будет полностью завершена.

Еще одним интересным моментом является операция присваивания. Какова будет ее семантика при наличии параллельных объектов?

p1: Process

p1 := p

Теперь p и p1 указывают на один и тот же параллельный объект. Т.е. по сути нет большой разницы при присваивании с обычными ref типами.

a1: Actor

a1 := a // concurrent into ref

a := a1 // ref into concurrent

!!! НЕ ПОМНЮ !!!

Критическая секция или захват ресурсов. Если некоторая процедура имеет хотя бы один параметр типа concurrent, то любой вызов такой процедуры означает эксклюзивный захват объекта, который передается в процедуру и снятие такого захвата по выходу из процедуры. Если таких параметров несколько, то вызов происходит тогда и только тогда, когда все объекты были эксклюзивно захвачены. Например.

concurrent unit Fork // Это вилка – образ параллельного разделяемого ресурса

end

concurrent unit Philosopher /\* Это философ – процесс, для работы которого (есть) требуется несколько ресурсов – вилок\*/

eatSpagetti (left, right: Fork) is // Чтобы есть спагетти нужны две вилки

end

end

men: Array [Philosopher] is …

forks: Array [Fork] is …

check

men.count = forks.count or else men.count = 1 and then forks.count = 2

end

while pos in men.lower .. men.upper loop

if pos = men.lower then

men (pos). eatSpagetti (forks(count), forks (pos))

else

men (pos). eatSpagetti (forks(pos-1), forks (pos))

end

end

Введение

Относительный порядок следования единиц компиляции не влияет на видимость имен единиц и на семантику программы в целом. Это означает, что традиционное текстово-ориентированное правило «сначала описание, затем использование»

<Общие слова о языке  
Целевая ориентация  
Важнейшие особенности и свойства  
Структура текста>

Общая структура программы на языке SLang

Compilation : { CompilationUnit }

Программа (в синтаксических терминах compilation) состоит из произвольного количества единиц компиляции (compilation units)[[1]](#footnote-1).

Формы физических носителей единиц компиляции (дисковый файл, поток ввода, область оперативной памяти и т.д.) в определении языка не фиксируются.

Каждая единица компиляции должна полностью размещаться в пределах одного физического носителя.

Относительный порядок следования единиц компиляции не влияет на видимость имен единиц и на семантику программы в целом. Это означает, что традиционное текстово-ориентированное правило «сначала описание, затем использование» для единиц компиляции не действует.

CompilationUnit  
 : { UseDirective } ( AnonymousRoutine  
 | StandaloneRoutine  
 | UnitDeclaration )

В языке определены три вида единиц компиляции: неименованная процедура, объявление независимой подпрограммы и объявление контейнера.

Для целей явной спецификации зависимостей между единицами компиляции, на самом верхнем уровне программы могут задаваться директивы использования. (синтаксически подобные фразы use могут также входить в качестве составных частей в объявления контейнеров и подпрограмм.)

Если в программе имеется неименованная процедура, то выполнение всей программы заключается в последовательном выполнении ее операторов. В противном случае, выполнение программа может начатся с любой независимой процедурыы или с одной из процедур инициализации контейнера.

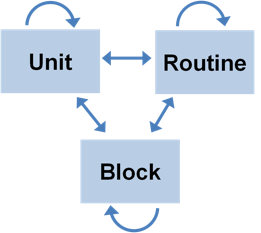
Поэтому конфигурационный файл определяет как начнется выполнение программы - либо как имя независимой процедуры, либо как имя контейнера в котором должна быть либо процедура инициализации без аргументов либо с одним аргументом массив строк, в котором будут передаваться аргументы командной строки.

Передача аргументов в стартовую процедуру реализуется посредством соответствующих системных вызовов или как единственный параметр типа массив строк (см. раздел X).

Вложенность компонентов программы

Важную роль в случае создания программных комплексов со сложной архитектурой играют правила вложенности компонентов. Практически все распространенные языки программирования накладывают те или иные ограничения на вложенность. Ярким примером может служить отсутствие вложенности функций в языке С и многих его потомках (на фоне этого запрета странно выглядит допущение локальных классов в С++).

Представляется, что общее правило должно быть максимально простым: любой компонент может содержать компоненты другого вида, в том числе и того же вида. Из этого правила следует возможность задания локальных и вложенных контейнеров и вложенных функций. Допустимые отношения вложенности могут быть наглядно представлены следующей схемой:



Представляется, что любое исключение из общих правил вложенности компонентов ограничивает возможности построения архитектуры ПО, адекватной требованиям, не продиктовано какими-либо концептуальными соображениями дизайна, но имеет реальной причиной желание упростить создание компилятора или других языковых утилит.

Неименованные процедуры

AnonymousRoutine  
 : StatementList

Неименованная процедурапредставляет собой последовательность операторов.

Если в программе имеется неименованная процедура, то она должна быть единственной в данной программе.

В предельном случае вся программа может состоять из единственной неименованной процедуры, то есть, из простой последовательности операторов (представлять собой своего рода «скрипт»).

Контейнеры: модули, классы и типы в одном флаконе

Важнейшими концепциями, используемыми при разработке программного обеспечения (ПО), служат понятия атрибутов (данных) и подпрограмм (действий). Атрибуты содержат данные и если они переменные, то могут изменяться подпрограммами в процессе работы программы; они образуют ее вычислительный контекст, в то время как подпрограммы задают алгоритм решения задачи. Между атрибутами и подпрограммами есть логические связи, и объединяя атрибуты и подпрограммы в единый именованный контейнер, мы просто фиксируем эту связь. Таким образом, понятие *контейнера* (английский термин, выбранный для его наименования,– unit) можно считать простым средством агрегации логически связанных данных и действий в единое целое.

Формальное определение контейнера звучит так. Контейнер (unit) – это поименованная совокупность атрибутов и подпрограмм, которая может быть параметризована либо типом, либо константным выражением перечислимого типа, и может быть использована для задания типов, конструирования новых контейнеров при помощи наследования или для прямого использования атрибутов и подпрограмм данного контейнера в других контейнерах и отдельно-стоящих подпрограммах.

Рассмотрим варианты использования контейнеров. Во-первых, его можно рассматривать как определение множества данных и операций над ними – то есть, как задание некоторого **типа**. Тем самым, можно определить объект, тип которого будет задаваться контейнером. Во-вторых, можно предоставить открытое (общедоступное) содержимое контейнера для **использования** в некотором программном коде, то есть, включить его ресурсы в некоторый контекст. Наконец, атрибуты и подпрограммы контейнера могут (пере)использоваться при создании нового контейнера. Такой механизм носит название **наследования**.

Таким образом, различные варианты использования контейнера приводят к понятиям *типа*, *модуля* и *класса*.

В большинстве современных языков программирования все перечисленные варианты композиции реализуются посредством единого понятия класса. Так, в С++ класс, все члены которого являются статическими, по существу, представляет собой простую агрегацию атрибутов и подпрограмм (вариант модуля). Аналогичное решение («статические классы») принято в языке C#. Заметим, что для представления модуля на основе класса приходится привлекать не вполне адекватное (обусловленное историческими причинами) понятие «статических» членов, а гибкое использование такого «модуля» (прежде всего, механизм его включения в определенный контекст) отсутствует.

В языке Scala [3] предпринята попытка отделить «модульную» часть класса в специальную конструкцию «объект-спутник» (companion object) с тем же именем, что и класс; однако на использование такого объекта накладываются существенные ограничения (в частности, он должен компилироваться в том же контексте, что и класс, спутником которого он является).

Определённым аналогом модуля можно считать механизм пространств имен (namespaces) в языках C++ и C#, а также пакеты (packages) языка Java, однако это крайне слабое средство модуляризации, введенное в языки прежде всего для разрешения конфликтов имен.

Таким образом, преимущества использования единственного понятия класса для задания различных видов программных контейнеров представляются сомнительными как с инженерной, так и с концептуальной точек зрения. В статье [1] с характерным названием «Импорт – это не наследование: Почему нам нужны и модули, и классы» приводятся убедительные обоснования необходимости явного разделения этих понятий. В то же время сосуществование в рамках единой языковой нотации понятий модуля и класса (например, в языках Ada [4] и Oberon [6]) выглядит несколько искусственным – при многих других несомненных достоинствах этих языков.

В данной работе предпринята попытка сохранить преимущества единой нотации задания контейнеров, совместив такую нотацию с явным заданием различных способов использования контейнеров.

Рассмотрим простой пример. Предположим, имеется некоторый контейнер, содержащий библиотеку математических функций.

unit Math is  
 sin(x: Real): Real  
 cos(x: Real) : Real  
 ...  
end

Наиболее логичное использование такого контейнера – это его использование как модуля: он должен явно включаться в клиентскую программу, в процессе ее работы он существует в единственном экземпляре, независимо от числа его использований. В то же время было бы желательно предоставить возможность переиспользования такого модуля, например, расширения его функциональности за счет включения новых математических функций. Такое изменение естественным образом реализуется посредством механизма наследования. Таким образом, контейнер должен обеспечивать как свойства модуля, так и свойства традиционного класса.

Итак, основную идею данного раздела можно сформулировать следующим образом: при сохранении единой формы задания контейнеров произвольной природы (модуль, класс, тип) обеспечить различные способы их использования. Иными словами, сделать так, чтобы один и тот же контейнер мог бы выступать в различных «ипостасях» в зависимости *от контекста его использования*.

Рассмотрим примеры, иллюстрирующие предложенную выше идею. Для упомянутого выше контейнера математических функций естественным (и, возможно, основным) способом использования будет включение его свойств в некоторый программный контекст:

use **Math**

Свойства Math (под *свойствами* контейнера мы будем понимать как его атрибуты, так и подпрограммы) будут использоваться традиционным способом: системой будет создан один объект (экземпляр) Math с собственным состоянием и предоставлять свой сервис всем контекстам, в которых явно задано использование этого юнита. Нотация доступа к свойствам Math допускает использование только квалифицированных имен, например, sin(x) или Math.Pi. НЕТ!!!! Договор был что всегда квалификация!!!

Наряду с этим, представляется естественным обеспечить возможность расширения функциональности Math – например, за счет включения новых функций или более эффективных реализаций имеющихся функций. Эта возможность реализуется традиционным для объектно-ориентированного подхода механизмом наследования:

**unit** BetterMath **extend** Math  
 ...  
**end**

Таким образом, в данном контексте контейнер Math используется как обычный класс.

Наконец, в некоторых обстоятельствах Math можно рассматривать как тип и, соответственно, использовать его при определении объектов, например:

m **is** Math

Такая запись вводит в текущий контекст объект с именем m, котрый инициализируется объектом типа Math, свойства которого определяются контейнером Math и доступны посредством обычной точечной нотации, например, m.sin(x).

Использование контейнера Math как типа допускает все возможности объектно-ориентированного, подхода, в частности, полиморфизм. Так, объекту m из примера выше можно присвоить объект производного типа и использовать «улучшенные» версии математических функций из этого производного типа:

m := BetterMath()  
...  
m.sin(x) // «Улучшенный» sin из контейнера BetterMath

Разумеется, природа конкретного контейнера не обязательно предполагает его реальное использование всеми тремя показанными способами. В каждом конкретном случае программист имеет возможность применить контейнер так, как требуется для решения конкретной задачи, в том числе и комбинируя способы, описанные выше. Кроме того, на тот или иной способ использования контейнера могут накладываться определенные ограничения. В частности, для того, чтобы контейнер мог использоваться как модуль, он должен иметь хотя бы одну процедуру инициализации («конструктор») без параметров или не иметь процедуры инициализации совсем.

Что это за текст красным цветом ниже????

У контейнера есть инвариант – он определяет ограничения на атрибуты контейнера. Проверка этого инварианта обеспечивает целостность контейнера.

Общее правило такое: если необходимо ввести какие-то дополнительные ограничения на атрибуты контейнера или на контейнер как таковой – вводи новый тип, наследуя его от имеющегося. Скажем, нужен тебе тип для положительных чисел (с соответствующим ограничением) – вводи тип Positive, наследуя его от Integer, и в этом новом типе Positive определяй собственный инвариант.

Вот как-то так...

Категории объектов и их области действия. Фраза use

В языке определены следующие пять категорий объектов в зависимости от их области действия и времени жизни:

- Предопределенные объекты  
- Глобальные объекты  
- Агрегируемые объекты  
- Объекты контейнеров  
- Локальные объекты

**Предопределенные объекты  
System-defined objects Я БЫ ЭТО ВЫБРОСИЛ!!!! Давай согласуем что это и зачем !!! В моем понимании – это объекты операционной системы и к языку не имеют никакого отношения!!!**

Объекты, определенные вне программы (на уровне операционного окружения или системы поддержки времени выполнения) и доступные для использования в программе У нас все доступно только на чтение!!!

Время жизни предопределенных объектов заведомо превышает время выполнения отдельной программы. Можно считать, что такие объекты «живут вечно».

***Замечание***. Это не означает, впрочем, что *значения* таких объектов сохраняются, и одна программа может воспользоваться значением, оставленным в предопределенном объекте некоторой другой программой. Следует исходить из того, что предопределенному объекту непосредственно перед запуском программы устанавливается некоторое конкретное значение.

<Привести примеры предопределенных объектов>

***Непонятка***. Если подобные объекты определены в некотором (системном, предопределенном) контейнере, то чем такие объекты отличаются от глобальных объектов?- у нас же все заключено в контейнеры, и, очевидно, будут «системные» (предопределенные) контейнеры?

**Глобальные объекты  
Assembly-wide objects**

НЕПРАВИЛЬНО!!!! Директива use на самом верхнем уровне говорит о том что создается один объект типа модуль – синглетон, а не куча объектов … Переписывай!!! Синглетон имеет внутреннюю структуру но он ОДИН!

Объекты, определенные в некотором контейнере и явно внесенные в глобальный контекст посредством объявления **use**, например:

**unit** A **is  
 ...  
end  
  
unit** X **is** global : A  
**end**  
...  
**use** X  
 // Объект X.global становится доступным в любой точке  
 // данного исходного файла программы посредством  
 // конструкции X.global

Такие объекты считаются доступными в любой точке данного исходного файла. Время жизни таких объектов совпадает с временем выполнения программы. Иными словами, глобальные объекты создаются в некоторый неопределенный момент времени до начала выполнения программы (***вариант***: при первом обращении к ним) и разрушаются после завершения ее выполнения.

Семантика конструкции **use** X заключается в том, что в данном исходном тексте программы будет доступен единичный объект-контейнер X («синглтон»), и все публичные имена из этого контейнера становятся доступными.

**Агрегируемые объекты  
Hierarchy-wide objects**

Если имя контейнера явно задано в заголовке некоторого другого контейнера в фразе **use**, то публичные ресурсы указанного контейнера становятся доступными внутри объявляемого контейнера. Пример:

**unit** X **is**  
 object **is** Integer /\* М используем тот факт что unit Integer имеет процедуру инициализации безх аргументов, которая задает значение 0\*/  
 set(i:Integer) **is** object := object + i **end**  
**end**  
  
**unit** A **use** X **is**  
 // Объект object и подпрограмма set из контейнера X  
 // становится доступны внутри объектов типа А  
 // и его потомков.  
**end**

Несколько более подробно и точно: фраза **use** X в заголовке контейнера A говорит о том, что в памяти будет создан объект-синглтон X, и объекты типа A получат доступ к данному синглтону. При этом все объекты типа A ссылаются на единственный экземпляр-синглтон контейнера X. Аналогичная фраза в заголовке некоторого другого контейнера приведет к созданию другого синглтона данного типа, который не будет связан с предыдущим.

***Замечание***. Необходимо подчеркнуть, что фраза **use** не подразумевает наследование: контейнер A из примера выше не наследует контейнер X. Он просто использует ресурсы X.

Аналогичным образом, фраза **use** в заголовке некоторой независимой подпрограммы (объявленной вне контейнера) приводит к образованию объекта-синглтона. В этом случае при любом вызове данной подпрограммы ссылки на атрибуты контейнера будут относиться к единственному объекту-синглтону. Тем самым, контейнер, заданный в заголовке такой функции, может служит своего рода «посредником» или хранителем информации, которая может передаваться от одного вызова подпрограммы к другому.

Пример:

foo(i: Integer) **use** X **is**  
 X.set(i)  
**end**

foo(1) // X.object равен 1  
...  
foo(100) // X.object равен 101

Таким образом, конструкция вида **use** X может представлять собой либо объявление на самом внешнем уровне исходного текста (как показано в предыдущем подразделе – вне объявлений контейнеров или подпрограмм), либо быть частью частью заголовка контейнера, либо частью заголовка standalone-подпрограммы (см. ниже). Фразу **use** нельзя использовать внутри контейнеров, подпрограмм или составных операторов.

***Замечание 1***. Вообще говоря, конструкция вида **use** X не является обязательной. Если в контейнере или в standalone-подпрограмме используется квалифицированное имя вида X.object без указания имени X в фразе **use**, то компилятор попытается идентифицировать имя X как имя некоторого контейнера (найдя его объявление в программе или его упоминание в конфигурационном файле). Если идентификация успешна, то описанная выше семантика **use** реализуется по умолчанию для ближайшего к месту использования контекста (то есть, либо для standalone-процедуры, либо для контейнера, в пределах которого встретилось квалифицированное имя).

Следует специально отметить, что наличие фразы **use** не означает, что атрибуты указанного контейнера будут доступен по своим коротким именам. Общее правило заключается в том, что ресурсы контейнера всегда должны квалифицироваться именем этого контейнера. Представляется, что такая практика будет приводить к большей читабельности и понятности текста программы и приведет к недопущению случайных конфликотов имен.

***Замечание 2***. Связывание имени контейнера, заданного в фразе **use**, с расположением этого контейнера (либо его исходного текста, либо кодового образа) на физическом носителе, производится вне программы (посредством конфигурационного файла).

Помимо указания на использование контейнера в укачестве поставщика ресурсов, фраза **use** может использоваться для задания синонима для используемого контейнера, например:

**unit** VeryVeryLongNameForTheUnit **is**  
 ...  
**end**

**unit** A  
 **use** VeryVeryLongNameForTheUnit **as** Short  
**is**  
 ...Short.object...  
**end**

**Объекты контейнеров (атрибуты)  
In-unit objects**

В эту категорию попадают все объекты, объявленные как члены контейнера. Их область действия ограничена самим контейнером. В то же время механизм доступа к публичным членам контейнера определяется способом использования контейнера (*это принципиальное положение дизайна языка*). Если контейнер используется как тип, например, в следующем объявлении:

x: X

то объекты из контейнера X доступны посредством их квалификации именем объекта этого типа, например:

a: Integer is x.object

Если же контейнер используется как модуль (одиночный объект, «синглтон»), то его публичные члены доступны посредством квалификации именем самого контейнера (об этом говорилось в предыдущем подразделе), например:

**use** X  
...  
foo(): Integer **is**  
 ...  
 X.object := 77 /\* Плохой пример !!! Он работает лишь тогда когда определен сеттер для атрибута object!!!! Перепиши плиз!\*/  
**end**

**Локальные объекты  
Local objects**

Локальные объекты объявляются внутри областей действия подпрограмм или блоков (то есть, внутри составных операторов). Доступность локальных объектов ограничено временем жизни блока, в котором такой объект объявлен.

Пример:

foo(): Integer **is**  
 local1: Real // объект local1 определен внутри  
 ... // тела функции foo  
 **if** local1 > 0 **then**  
 local2: Real // объект local2 определен внутри  
 ... // then-части оператора if  
 **end** // Здесь объект local1 существует;  
 // объекта local2 уже не доступен и может быть удален  
 ...  
**end**// Здесь объекта local1 уже не доступен!!!

**Локальные объекты и области действия имен**

To be discussed…

Тело подпрограммы, а также все составные операторы образуют области действия. Это означает, что имя объекта, объявленное в пределах области действия, действительно от точки его объявления до конца соответствующей области действия.

С другой стороны, в языке не допускается перекрытие имен, то есть, имя объекта, введенное в некоторой области действия, не должно совпадать с именем из объемлющей области действия.

Представляется, что такое ограничение, будучи несколько более жестким, нежели в традиционных языках, будет способствовать большей понятности и читабельности программ. Заметим, что подобное ограничение действует и в языке Java.

Пример:

foo(): Integer **is**  
 local: Real // объект local определен внутри  
 ... // тела функции foo  
 **if** local > 0 **then**  
 local: Real // Ошибка времени компиляции:  
 // повторное объявление  
 **end  
end**

**Простые объявления  
Simple declarations**

Начнем с самых простых примеров. Вот первый:

x **is** 5

Это весьма компактное, но вместе с тем полноценное объявление-инициализация, в котором присутствует вся необходимая информация для построения компилятором объектного кода и для контроля за последующим использованием объекта x. Кроме того, это объявление выглядит интуитивно понятным без дополнительных объяснений.

Более конкретно, это объявление говорит, что в программе возникает ***объект***, последующая работа с которым будет производиться по его имени x. Этот объект в процессе работы программы будет содержать значение ***целочисленного типа***. Это обстоятельство выявляется компилятором из типа ***начального значения***, которое получает объект x при его объявлении. Таким образом, значение (целочисленный литерал) 5 играет в данном случае двоякую роль: на его основе выводится тип объекта x и число 5 становится значением («начальным значением») объекта x.

Здесь необходимо сделать два ***замечания*** относительно синтаксиса этого объявления. Во-первых, мы используем служебное слово **is** для связывания имени объекта с его значением. Это служебное слово служит индикатором конструкции объявления-инициализации. Если использовать для этих целей, например, знак присваивания, то конструкция становится неотличимой от обычного оператора присваивания вида x := 5. Тогда пришлось бы вводить специальное служебное слово, чтобы показать компилятору, что перед ним именно объявление.

Второе замечание носит более общий характер и касается использования специальных знаков для обозначения концов тех или иных конструкций. Обычно для этих целей используются точки с запятой. Однако, в большинстве случае символы-завершители являются с синтаксической точки зрения избыточными и служат только для повышения наглядности программ.

В языке принято общее решение, согласно которому символ, завершающий конструкцию (в данном примере объявление) может быть опущен. Более подробное описание правил (не)задания завершителей будет приведено в разделе X.

Как уже было сказано, в объявлении задаются два свойства переменной: её тип и начальное значение. Начальное значение задаётся явно, а тип выводится компилятором из типа заданного значения. При необходимости или при желании тип объявляемого объекта может быть задан и явно, например:

x: Integer **is** 5

Здесь компилятору нет необходимости производить анализ инициализирующего выражения, чтобы определить тип, назначаемый объекту. В общем случае тип заданный для атрибута при описании может отличаться от типа инициализирующего выражения при этом либо тип инициализирующего выражения конформен типу в описании либо существует операция присваивания в юните, которая позволяет конвертировать объекта типа инициализирующего выражения в тип атрибута.

**Объявления: тип и начальное значение**

Зададимся вопросом: а можно ли объявить объект, задав ему только тип? Рассмотрим следующий пример объявления:

x: Integer // ?

С этого примера начнем обсуждение одного из базовых принципов дизайна языка SLang. Этот принцип состоит в том, что вообще говоря, в программе **не может существовать объектов, не имеющих значения[[2]](#footnote-2)**. Этот подход (детально он будет обсуждаться ниже) позволяет избежать многочисленных, разнообразных и трудноуловимых ошибок в программах. Поэтому в общем случае при объявлении объекта необходимо задать ему некоторое начальное значение.

Вообще говоря, имеется три основных способа задания значения объекту:

- Указать начальное значение непосредственно при его объявлении  
- Задать начальное значение неявно  
- Инициализировать объект посредством некоторой процедуры инициализации.

Обсудим эти три варианта подробно в применении к различным контекстам. Но сначала введем общее правило. Оно вытекает из предыдущего правила («все объекты должны иметь значение») и звучит достаточно просто: **все объекты в программе должны быть проинициализированы – явно или неявно**.

Из этих правил, правда, есть одно исключение, которое будет рассматриваться далее в отдельном подразделе.

Первый, самый простой и наглядный способ инициализации был представлен в начальном примере. Объект x при объявлении получает тип и начальное значение. Посредством таких объявлений можно вводить как локальные объекты в подпрограммах, так и атрибуты контейнеров, например:

**unit** A **is**  
 x **is** 5  
**end**

foo() **is**  
 x **is** 5  
**end**

Дальше начинаются отличия. Рассмотрим пример объявления атрибута контейнера:

**unit** A **is**  
 x: T  
**end**

Здесь инициализация атрибута явно не задана. Такая форма объявления считается допустимой только в случае, когда в процедуре инициализации контейнера имеется явное присваивание атрибуту x некоторого (начального) значения, например:

**unit** A **is**  
 x: T  
 **init** **is** x := T(1) **end**  
**end**

Альтернативой присваиванию в процедуре инициализации служит явное присваивание, например:

**unit** A **is**  
 x: T **is** T(1)  
**end**

В обоих приведенных примерах реальная инициализация атрибута x произойдет при создании экземпляра контейнера, точнее, при выполнении его процедуры инициализации.

В отличие от объявлений атрибутов контейнеров, локальные объекты следует объявлять только с инциализацией. Иными словами, в их объявлениях должна обязательно присутствовать фраза **is**, например:

foo() **is**  
 x1**:** T // ошибка: не задано начальное значение  
 x2: T **is** T(4) // правильно  
 x3 **is** T(7) // тоже правильно  
**end**

====================================

Запись вида x **is** 5 является более короткой формой объявления

x **is** Integer(5)

Значением объекта x становится объект типа Integer, который возникает в результате работы инициализатора с одним параметром.

a: Integer  
b: T **is** 5 // начальное значение b – вызов  
 // процедуры инициализации T(5).  
b: T **is** T(5) // Полный вариант предыдущего объявления

Таким образом, в распоряжении программиста имеется несколько вариантов записи простого объявления, например:

y **is** 77 // самая короткая форма  
y **is** Integer(77) // два частичных  
y: Integer **is** 77 // сокращения  
y: Integer **is** Integer(77) // полная форма  
y: Integer **is** Integer.**init**(77) // самая полная форма ☺

Описанный подход к объявлению применим к объектам любых типов. Вот, например, как выглядит объявление массива:

a **is** (1, 2, 3, 4, 5)

Конструкция после служебного слова is представляет собой ***изображение массива***. Эта запись содержит всю необходимую информацию об объявляемом массиве: скобки служат для группирования нескольких однородных значений в единую конструкцию (эта конструкция в данном контексте представляет собой изображение массива), элементы в скобках (в общем случае это могут быть произвольные выражения), становятся элементами массива; их количество задает размер массива, а тип выражений (в данном случае он должен быть одним и тем же для каждого элемента списка или если быть совсем точным, то все типы выражений должны быть конформны некоторому типу Т, который и будет типом элементов массива) определяет тип элементов массива.

***Замечание***. На самом деле, и это будет показано в последующих разделах, конструкция со списком элементов, заключенных в скобки, носит более универсальный характер. Вообще говоря, любой подобный список называется в языке кортежем (tuple) и может представлять различные сущности. Однако в контексте объявления без явного задания типа список трактуется как изображение массива.

Таким образом, объект a из объявления выше получает тип, который может быть явно записан как Array[Integer]. Как видно из формы задания массивов, предопределенный тип Array представляет собой обобщенный («generic») класс, типовой параметр которого задает тип элементов.

При необходимости можно явно указать тип....

**Кортежи  
Tuples**

**1. Общее описание**

Многие синтаксические конструкции в различных языках программирования включают повторяющиеся элементы (списки, последовательности, наборы) однородных и/или разнородных элементов. Примерами могут служить изображения массивов и других составных структур, списки идентификаторов в объявлениях, списки параметров и аргументов функций, последовательности описания членов классов и многое другое. Определение перечислимых типов в некоторых языка также можно считать примером однородных последовательностей.

Семантика подобных последовательностей может быть различной, однако представляется разумным и перспективным выделить общие аспекты таких конструкций и попытаться их обобщить в рамках единой языковой нотации.

Итак, введем понятие **кортежа** (**tuple**) как произвольной совокупности некоторых языковых элементов («tuple is a group of something») и рассмотрим контексты, в которых это понятие используется или могло бы использоваться.

Вообще говоря, кортеж полезен и должен быть применим всюду, где по смыслу программы необходимо иметь дело с несколькими объектами как с единым целым. По причине сильной близости кортежей к привычным конструкциям ЯП, их синтаксис должен быть компактным, узнаваемым и очевидным и не должен вызывать удивления, а свобода формирования кортежей и манипуляций с ними должна быть максимальной.

Во многих ситуациях группирование нескольких сущностей выглядит естественным. Так, вызов функции с передачей ей нескольких аргументов представляется очевидным примером такого группирования:

foo(a,1+x,**true**)

Изображение массива в языках С/С++ также может служить примером кортежа:

**int** A[10] = { 1, 2, 3, 4, 5 };

Спецификация функции представляет пример совокупности сущностей, представляющих либо полные объявления параметров вида «имя-тип»:

foo(p1:Integer, p2:Real, p2:Boolean): Real

либо просто совокупность типов, которую в данном случае можно трактовать как список неименованных параметров:

foo(Integer, Real, Boolean): Real

В принципе, две последние формы кортежей можно обобщить и считать, что совокупность именованных и/или неименованных объявлений представляет собой некоторую структуру – своего рода «анонимный класс», или специальный вид **составного типа данных**.

Для такого типа можно определить различные контексты использования и операции, в частности:

Объявления объектов типа «кортеж»

x: (Integer, Real)

Переменная x представляет пару значений, первое из которых имеет целочисленный тип, второе – вещественный тип.

y : (a: Integer, b: Real)

Это объявление аналогично предыдущему, но здесь элементы кортежа (и вместе с ним, элементы переменной y) поименованы.

В принципе, нет никаких препятствий для «смешанной» записи, когда поименованы только некоторые некоторые элементы кортежа, например:

z: (a: Integer, Real, c: Boolean)

Будучи объявленной, переменная типа кортеж может допускать полный набор операций, прежде всего, присваивание (включая передачу в качестве аргумента в функцию и возврат из функции в качестве результата), а также доступ к отдельным элементам кортежа.

Операции доступа к элементам возможны двумя способами: либо по порядковому номеру элемента в кортеже, либо по имени элемента (если он задан). Доступ по порядковому номеру возможен всегда, независимо от того, поименованы ли элементы кортежа (нумерация элементов начинается с единицы); доступ по имени возможен в случае, когда соответствующий элемент кортежа имеет имя.

А Баба-Яга против!!! Почему такое ограничение для общего случая? Почему в группу нельзщя добавить новый элемент или убрать существующий? Чему это противоречит?

Для доступа к поименованным элементам кортежа используется традиционная точечная нотация, в которой задается имя всего кортежа и имя элемента, к которому производится доступ. Для неименованных элементов используется запись, традиционная для агрегатных сущностей (например, массивов), где порядковый номер элемента («индекс») задается в скобках после имени кортежа.

x := (1, 2.3) // присваивание кортежу в целом  
x(1) := 7 // доступ к первому элементу кортежа

Пример выше плох тем что сразу надо давать объяснение что кортеж имеет оператор := переопределенным. Лучше оставить пример вида где как раз доступ а не записть в элемент кортежа …

var **is** x (1) /\* Специально для тебя назвал переменную var :-) \*/  
  
b: Real **is** y.b // объявление вещ. переменной  
 // с инициализацией элементом кортежа  
 **if** z.c **then** /\* do something \*/ **end**

Заметим, что индексная форма записи, как более универсальная, может применяться и для доступа к именованным элементам кортежей, например:

z.c := **false** // два эквивалентных способа  
z(3) := **false** // доступа к элементу c кортежа z  
ПЛИЗ замени примеры с записью на примеры с чтением из кортежа …

s **is** 2  
...  
z(s\*3-5) := 7.125 // номер элемента задан выражением

В заключение приведем несколько более развернутый пример. Пусть имеется следующая конфигурация контейнеров:

**unit** Base **is**  
 foo() is ... **end**  
**end**  
**unit** X **extend** Base **is**  
 **override** foo() **is** ... **end**  
**end**  
**unit** Y **extend** Base **is**  
 **override** foo() **is** ... **end**  
**end**

Тогда работу с кортежем, составленным из объектов этих контейнеров, можно организовать следующим образом:

k: (b:B, x:X, y: Y)

**while** i **in** 1..k.count () **loop**  
 k(i).foo()  
**end**

или еще проще: А вот тут не уверен !!! Тип elem – какой??? По-моему Any а в Any нет рутины foo …

**while** elem **in** k **loop**  
 elem.foo()  
end

ПЛИЗ у нас один цикл loop и условие while модет быть либо впереди либо в конце …

**2. Кортежи и функции**

Понятие кортежа в том виде, в котором оно введено выше, хорошо согласуется с механизмом параметров подпрограмм и с заданием возвращаемых значений. Так, список параметров подпрограммы можно трактовать как кортеж, например:

foo(p1: Real, p2: Integer, Real): Real **is** ... **end**

Если необходимо определить подпрограмму, возвращающую несколько значений, то результирующий тип может быть естественно задан в виде кортежа:

squareRoots(a: Real, b: Real, c: Real) : (Real, Real)  
**is**  
 ...  
**end**

roots **is** squareRoots(2,3,7)  
root1: Real **is** squareRoots(1,2,3)(1)

Заметим, что возможность задания нескольких результирующих значений во многих реальных случаях способствует более ясному дизайну подпрограмм, избавляя от необходимости использовать другие, менее наглядные и менее надежные способы возврата значений.

Предельным вариантом кортежа естественно считать кортеж, не содержащий ни одного элемента. Его можно обозначить с помощью пустых круглых скобок.

***Замечание***. Есть большое искушение (и кажется естественным) использовать пустой кортеж для задания подпрограмм без параметров и подпрограмм, не возвращающих значение. Однако по ряду причин (объяснение будет дано в одном из следующих подразделов данной главы) такая форма в языке не допускается. Подпрограмма без параметров должна задаваться следующим образом:

strange : Real **is** ... **end**

ПОЧЕМУ?????

Function (): Type **is** … **end**

Что тут плохого????

Подпрограмма, не возвращающая значения, будет иметь следующий вид:

strange2(p: Real) **is** ... **end**

Заметим, что такая форма приводит к некоторой (правда, не слишком вопиющей) синтаксической неоднозначности, когда такая, например, конструкция

strange3 **is** *Expression* **end**

ЭТО СИНТАКСИЧЕСКАЯ ОШИБКА!!! ЗДЕСЬ НЕТ НЕОДНОЗНАЧНОСТИ!!!!

Это не есть процедуда так как стоит выражение последним перед end и значит что перед ним просто пропущен return и значит это функция, а она по виду процедура -= в теле процедуры не может стоять выражение!!! ЭТО НЕ СИ !!!

может идентифицироваться как объявление подрограммы без параметров и без возвращаемого значения (а не объявление переменной с инициализацией) только при обнаружении завершающего **end**.

Синтаксис и семантика объявлений подпрограмм будут представлены в отдельном разделе.

У НАС НЕ СОГЛАСОВАНО ТАКОЕ КЛЮЧЕВОЕ СЛОВО!!! Я бы делал это по-другому – как писал уже не раз

arguments **is** foo.arguments

**while** argument **in** arguments **loop**

**//** Тип переменной argument is Any!!!

**end**

В списке параметров допускается задание неименованных элементов; это также распространенная практика программирования. В этом случае обращение к неименованным параметрам в теле подпрограммы может производиться посредством специального служебного слова **param**, например:

foo(p1: Real, Integer, Real): Real **is** ...  
 **param**(1) *или* **param**.p1 *или* p1  
 **param**(2)  
 **param**(3)  
 ...  
 **end**

С привлечением понятия кортежа так же естественно решается вопрос с умолчаниями для параметров:

goo(p1: Real, Integer, p3 **is** 0, **true**): Real **is** ... **end**

В этом примере первый параметр – обязательный именованный. Обращение к нему производится по имени. Второй параметр имеет тип Integer; он не имеет имени, но при вызове является обязательным. Обращаться к нему можно посредством конструкции **param**(2). Третий параметр является необязательным; при вызове функции его можно опускать, в этом случае он принимает значение по умолчанию 0. Обращение к нему внутри тела подпрограммы может производиться как по имени p3, так и с помощью конструкции **param**(3). Наконец, последний, четвертый параметр необязательный (в случае, когда он опущен, его значение есть **true**) и имени не имеет. Обращение к нему возможно только посредством **param**(4).

Тип последних двух параметров явно не задается и выводится компилятором из типа значений по умолчанию (Integer и Boolean).

Да, я это придумал (обращение через **param**). Умолчания для параметров – повсеместная практика, есть во всех языках. Выведение типов непосредственно следует из принятого дизайна языка. Так что... все логично.

МНЕ param НЕ НРАВИТЬСЯ – опять тратить ключевое слово на совсем редкую вещь … Просто от имени функции возьми апарметры … Функция это объект у него есть тип – юнит в нем есмть некотрые свойства …

**3. Пустые и единичные кортежи**

Как уже говорилось, предельным вариантом кортежа естественно считать кортеж, не содержащий ни одного элемента. Его можно обозначить с помощью пустых круглых скобок. Подробнее о пустых кортежах см. след. подраздел.

Некоторая неопределенность возникает при определении единичного кортежа – который включает единственный элемент. Дело в том, что в этом случае в некоторых контекстах невозможно отличить единчный кортеж от простого выражения, заключенного в круглые скобки.

Пример:

x **is** (*Expression*) // x – это кортеж или простой объект?

Для случаев, когда может возникнуть подобная неопределенность, принято следующее правило: если в скобках содержится одиночное выражение, оно всегда трактуется как выражение, а не как кортеж из одного элемента. Выбор такой трактовки обусловлен тем обстоятельством, что выражения в скобках (как отдельно-стоящие, так и входящие в состав других выражений) гораздо чаще встречается в реальных программах, нежели единичные кортежи.

Если же необходимо задать кортеж из одного элемента, то для этого следует явно указать тип объекта как кортеж. Так, предыдущий пример может быть переписан, например, следующим образом:

x: (Real) **is** (*Expression*) // x – кортеж из одного  
 // вещественного элемента

**4. Конформность между кортежами. Кортежи, контейнеры и массивы**

Считается, что кортеж из N элементов определенных типов конформен любому другому кортежу из N+k элементов, первые N типов которого попарно конформны соответствующим типам первого кортежа.

НЕПРАВИЛЬНО !!! Все ровно наоборот!!! Кортеж с числом элементов N+K конформен кортежу с N элементов тогда и только тогда когда все N первых элементов кортежа N+K конформны соответствующим N элементам второго кортежа.

Например:

x: (Real, Integer, Boolean) is (1.0, 777, false, "эта строка не попадает в x")

Из этого правила следует, что пустому кортежу конформен любой другой кортеж.

Именно из-за этого следствия из общего правила конформности использовать пустые кортежи для обозначения отсутствия параметров в подпрограммах и для отсутствия возвращаемого значения было бы неправильно. В самом деле, если допустить форму вида

foo() : Real **is** ... **end**

то согласно правилу конформности кортежей все следующие формы вызова такой подпрограммы оказались бы корректными:

И ОНИ КОРРЕКТНЫ!!!!! Именно при правильном определении комфортности кортежей!!! ПЕРЕПИСЫВАЙ!!!!

foo()  
foo(1)  
foo(7.2, **true**,(1,2,3))  
...

Аналогичные рассуждения приводят также к выводу о недопустимости задания пустого кортежа для обозначения отсутствия возвращаемого значения.

ПУСТОЙ КОРТЕЖ НЕ ЕСТЬ ОТСУТСВИЕ ЗНАЧЕНИЯ – ПУСТОЙ КОРТЕЖ И ЕСТЬ ЗНАЧЕНИЕ!!!! А динамически тут может быть любой кортеж!!!

Ранее уже говорилось, что совокупность элементов, сгруппированных в кортеж, в некотором отношении подобна простому контейнеру, содержащему аналогичные элементы в качестве своих атрибутов. Иными словами, можно трактовать кортеж как изображение некоторого «анонимного» контейнера. А если так, то имеет смысл установить правило конформности между контейнерами и кортежами.

А ВОТ ЭТОГО НЕ НАДО!!! Если кто-то хочет задать := для некоторого юнита с аргументом типа кортеж – плиз а вот вводить БЕЗ МЕНЯ новые правила комфортности ☺

Итак, пусть имеется контейнер с атрибутами, типы которых обозначим как ***T1***, ..., ***Tn***. Тогда кортеж с ***n+k*** элементами, в котором первые ***n*** элементов попарно конформны соответствующим элементам атрибутов контейнера, считается конформным данному контейнеру.

Из этого правила следует, в частности, допустимость «группового» присваивания объектам-контейнерам конформных ему кортежей, или аналогичной инициализации, например:

**unit** A **is**  
 a: Real  
 b: Integer  
 c: Boolean  
**end**

a: A **is** (1.2, 777, **true**)

ОСТЬАНОВСИЬ !!! В ОБЩЕМ СЛУЧАЕ ТО ЧТО ТЫ ПИШЕШЬ НЕПРАВИЛЬНО!!! Вспомни про наследование – ты же не знаешь всех полей а если я еще три поля пронаследовал, да еще пару переопределил то в каком порядке надо проверять конформность а ведь есть еще и множественное наследование. НУ КОГДА ТЫ ЗАБУДЕШЬ ПРО С++ ☺ Layout юнита не определяется текстом программы !!!!!

Если я образаюсь к юниту как к кортежу и говорю ему дай мне твой первый эелемент то что я получу – то что по алфавиту превое или в тексте а в какм юните базовом или производном – выкини С++ из головы когда думаешь про SLang! ☺

Продолжая эти рассуждения дальше, можно распространить их на другие агрегатные структуры, например, на массивы. В самом деле, любой массив можно представить как кортеж, все элементы которого имеют один и тот же тип. Из этого обстоятельтства немедленно следует вывод о том, что кортеж с любым числом элементов, имеющих один и тот же тип **T**, конформен массиву элементов того же типа **T**. Пример:

a: Array[Real] **is** (1.2, 3.4, 5.6, 7.8, 9.0)

В точности те же рассуждения можно применить и к спискам:

НЕ РАБОТАЕТ по твоей логике. А работем по моей если в юните List я определил := с параметров Array [Any] то вот такой пример будет работать – ТАК И НИКАК ИНАЧЕ ! ☺

lst: List[Integer] **is** (1, 2, 3, 4, 5, 6, 7, 8, 9)

**5. Инварианты для кортежей**

Если уж мы «уподобили» в некотором смысле кортежи контейнерам, то представляется естественным распространить на кортежи свойства, присущие контейнерам,– в частности, понятие ***инварианта***. В самом деле, выглядит вполне естественным, если программист захочет установить некоторые требуемые перманентные соотношения между элементами кортежа, которые должны выполняться в течение его жизненного цикла.

Рассмотрим несколько примеров. Следующий кортеж не содержит инвариант.

t **is** (f1: Integer; f2: Real; f3: Boolean)

Отсутствие инварианта можно трактовать как допущение любых значений его элементов или, что то же самое, наличие в данном кортеже неявного инварианта, тождественно возвращающего значение **true**.

После последнего элемента кортежа допускается задание, после служебного слова **invariant** (аналогично контейнерам), логического выражения над его элементами:

t **is** (f1: Integer; f2: Real; f3: Boolean **invariant** f1>=f2=>f3)

***Непонятка***: А можно задавать инварианты для... параметров подпрограмм?? Вроде, у нас параметры – тоже кортеж, не так ли? И необходимость, вроде, есть... Можно возразить: есть же предусловия для подпрограмм. Отвечаю: предусловия предусловиями, они обозначают *ограничения* на каждый отдельный параметр – а инвариант задает *отношение* *между* параметрами...

***Для подпрограмм инварианта как понятия нет в общем случае - есть предусловие и постусловие и есть инвариант контекста в котором происходит вызов подпрограммы …***

В данном примере кортеж определяется как совокупность трех элементов, связанных заданным в конце логическим выражением (операция => обозначает импликацию, то есть логическое следование). Это выражение должно соблюдаться для любого состояния кортежа; иными словами, его истинность будет проверяться после любого изменения значения какого-либо элемента кортежа.

Так, начальное значение данного кортежа после исполнения объявления представляет собой совокупность следующих значений: (0, 0.0, **false**), что следует из инициализаторов для типов соответствующих элементов кортежа. Эти значения, как можно видеть, удовлетворяют условию из инварианта.

Присваивание

t.f1 := 5 // illegal: invariant violated

недопустимо, так как в результате его выполнения истинность инварианта будет нарушена. С другой стороны, присваивания

t := (5, 1.0, **true**) // OK: invariant preserved  
t(2) := 4.99 // OK: invariant preserved!

корректны, так как не нарушают инвариант.

**6. Кортежи и переменное число аргументов подпрограмм**

<Проблема до конца не решена>

В чем затык????

StandardIO.put (1, true, “String”, ‘C’, 5.5, (1,2,3))

**unit** StandardIO

put (arguments: ()) **is**

**while** argumentPos **in** 1 **..** arguments.count **loop**

argument **is** arguments (argumentPos)

**case** argument

**when** Integer **then**

putInteger (argument)

**when** Character **then**

putCharacter (argument)

**when** String **then**

putString (argument)

**when** Real **then**

putReal (argument)

**when** Boolean **then**

putBoolean (argument)

**else**

put String(argument.ToString)

**end**

**end**

**end** put

**hidden external** putInteger (anInteger: Integer)

**hidden external** putCharacter (aCharacter: Character)

**hidden external** putString (aString: String)

**hidden external** putReal (aReal: Real)

**hidden external** putBoolean (aBoolean: Boolean)

**end** StandardIO

* Then every routine has only 1 parameter – tuple, possibly empty. And it returns a tuple with 0 or more elements. Procedure is a function which returns empty tuple ☺ So, we can just ignore what it returns like void in old plain C ☺

**Подпрограммы  
Routines**

**Исключительные ситуации  
Exceptions**

Семантика исключительных ситуаций в языке SLang в целом соответствует аналогичным механизмам в других современных ЯП. Возникновение («возбуждение») исключительной ситуации приводит к немедленному прекращению выполнения блока, в пределах которого она возникла.

Под «блоком» здесь понимается либо тело некоторого составного оператора, либо тело подпрограммы.

Нет у нас составного оператора … Это не Паскаль, не С и не 1970 год ☺  
Условные, циклы – это **составные** операторы: в их **состав** входят другие операторы. Это любой современный язык. 2016 год.

Если в блоке, в котором возникла ситуация, предусмотрен обработчик соответствующей ситуации, то управление передается этому обработчику. В этом случае ситуация считается перехваченной. В противном случае производится передача управления в блок, динамически объемлющий данный блок. Если в нем предусмотрен обработчик, то управление передается ему, в противном случае алгоритм повторяется до тех пор, пока не будет найден подходящий обработчик либо пока не будет обнаружен самый внешний блок программы – то есть, подпрограмма, с которой началось выполнение всей программы. В последнем случае программа завершается аварийно.

Современные ЯП различаются трактовкой природы ситуации как таковой, смысла «обработчика, соответствующего ситуации», а также философии использования всего механизма. Так, в языке Ада ситуации считаются специфическими сущностями, отличными от всех других сущностей программы. Различение ситуаций производится по их именам. С другой стороны, в таких языках, как C++/Java/C# семантика ситуаций встроена в общую объектно-ориентированную парадигму, согласно которой ситуация – это объект определенного типа, и различение ситуаций производится по их типам согласно общим отношениям наследования.

При возбуждении ситуации в Ада-программе поиск подходящего обработчика производится по совпадению его имени с именем возникшей ситуации. Вместе с ситуацией как таковой по динамической цепочке вызовов передаются ее *атрибуты*, среди которых программист может задать некоторую полезную информацию (например, информацию о месте и характере исключения). В С++ и «производных» от него языках исключение как таковое представляет собой объект некоторого типа. В качестве «типа ситуации» может использоваться либо произвольный – стандартный или пользовательский тип (С++), либо тип, производный от некоторого библиотечного типа (например, System.Exception в C#).

Наконец, языки различаются по подходу к использованию механизма исключений. Так, в языке Ада, а также в языке Эль-76 ситуации трактуются как один из возможных и допустимых механизмов организации потока управления. В системе Эльбрус (для которой проектировался язык Эль-76) ситуации поддерживались на аппаратном уровне, такой подход был относительно оправдан, так как его систематическое использование не приводило к заметному падению производительности. С другой стороны, аппаратная организация современных процессоров не обеспечивает должной степени эффективности исключений (как правило, реализация исключений основана на так называемых «длинных переходах» и сравнительно дорога в плане эффективности).

Другим негативным аспектом использования механизма исключений как средства организации потока управления является нарушение <слова об инвариантах, постусловиях и прочем>.

Таким образом, при проектировании механизма исключительных ситуаций языка SLang в его основу были положены следующие принципы:

- Данный механизм (согласно смыслу названия) является исключительным средством управления и должен использоваться в ситуациях, которые делают невозможным «нормальную» организацию выполнения согласно общим принципам структурного программирования.

- «Ситуация» представляет собой объект определенного типа. При возникновении исключительной ситуации этот объект передается обратно по динамической цепочке вызовов до тех пор, пока не будет обнаружен обработчик, в спецификации которого задан тот же тип, что и у ситуации, или которому конформентип текущего исключения. Если такой обработчик найден, ситуация считается перехваченной и управление передается обработчику. В противном случае поиск обработчика повторяется до тех пор пока не будет найден подходящий обработчик либо по не будет достигнут самый внешний блок прогораммы. В последнем случае программа завершается аварийно.

- Если найденный обработчик выполнился успешно, ситуация, которую этот обработчик перехватил, считается обработанной. Если в обработчике возникла некоторая ситуация, то механизм работает, как описано выше. Ситуация считается перехваченной, но не обработанной. Если в обработчике (повторно) возникла та же ситуация, которая была перехвачена, то механизм работает, как описано выше; при повторном возникновении ситуации вверх передается в точности тот же объект, который пришёл в обработчик.

- На тип ситуации не накладывается никаких ограничений. Тип может быть как пользовательским, так и произвольным предопределенным, в частности, Integer, Real и т.п.

**try**  
 *Statements*  
**catch** ( a: T1 )  
 *Statements*  
**catch** ( a: T2 )  
 *Statements*  
**end**

***Замечание***. В некоторых языках (например, в Java) функция, в которой могут возникнуть исключительные ситуации, в своем заголовке должна специфицировать все эти (возможные) ситуации. С одной стороны, это способствует более аккуратному программированию и допускает применение автоматических методов статического анализа корректности программ. С другой стороны, практика показывает, что такое правило существенно затрудняет программирование, ограничивает гибкость программ и не приносит существенных выгод в плане надежности. По этим причинам аналогичное правило в последних редакциях стандарта С++ отмечено как «устаревшее» (obsolete).

**Базовые юниты – система типов.**

Так как в языке SLang каждый юнита определяет тип, то встает вопрос о том, какие юниты (а следовательно и типы) являются базовыми или предопределенными. Основой системы типов являются два типа – данные в формате последовательности битов и понятие подпрограммы (функции). Или другими словами в нотации языка есть два юнита – Bit [N:Integer] и Routine [Arguments->(), Result], которые являются основой всего многообразия типов. А также постулируется существование юнита Any, который является непосредственным родителм любого юнита, у которого в явном виде отсутствует спецификация наследования, который содержит описание операций сравнения объектов и присваивания. Давайте разберем сначал описания этих юнитов. Юнит Bit [N:Integer] – это по сути абстракия битового поля, которая параметризуется целочисленной константой, которая задает количество битов в этом поле. Инвариантом данного юнита является утверждение, что N > 0, что вполне логично. Верхней границы на количетсво битов в поле накладывать не имеет смысла, достаточно мощности целого типа для такого ограничения. Две основные операции над битовым полем, которые не представляется возможным выразить на самом языке программирования – это взятие и соответственно установка конкретного бита в конкретной позиции битового поля. Все остальные операции можно выразаить через эти две операции. Вот как они задаются

**external** () (pos: Integer; value: Integer) /// Set bit as position 'pos' to value 'value'

**require**

pos **in** 0 .. N - 1 /// Valid bit position

value **in** 0..1 /// Valid value

**end** // ()

**external** () (pos: Integer): Integer /// Get bit value at position 'pos'

**require**

pos **in** 0 .. N - 1 /// Valid position

**ensure**

**return** **in** 0..1 /// Valid Integer from Bit

**end** // ()

Для того чтобы на примере рассмотерть как работают две операции, представленные выше рассмотрим следующий фрагмент кода.

b1: Bit 16 **is** 101b

b2: Bit 16 **is** 100b

b1(0) := b2 (1) // Установить 0-й бит перменной b1 в значение, взятое из 1-ого бита переменной b2

b1 (o, b2 (1)) // Таже самая операция только записанная в процедурном стиле

Следующий вопрос, который всегда возникает при проектировании юнита – это какие процедуры инициализации у него должны быть. Для юнита Bit [N: Integer] предусмотерна одна процедура инициализации без параметров – и тело ее содержит цикл который всем битам данного битового поля присваивает значение 0. Таким образом даже тело процедры иницализации не является чем-то особенным или внешним, а выражается через операцию установки бита в нужное значение.

1. Вырожденный случай, когда число таких единиц нулевое, считается допустимым и при этом порождается пустая программа которая ничего не делает. Или что???? [↑](#footnote-ref-1)
2. Если быть более точным, то объекты, не имеющие значений, существовать могут, но никаких операций, кроме присваивания им значения, над ними выполнять нельзя, и это проверяется статически при компиляции программы. [↑](#footnote-ref-2)