**Блок: базовая конструкция**

Базовая конструкция действия (или просто ***блок***) задается следующим образом:

**do**  
 *Объявления и операторы* **end**

Данная конструкция, помимо группирования объявлений и операторов в единую конструкцию, определяет собственный *локальный контекст* – область действия имен, объявленных в ее текстуальных пределах. Так, в следующем примере

**do**  
 x: Integer  
 x := 1  
 **end**

область действия объекта x начинается от точки его объявления и завершается непосредственно перед служебным словом **end**.

Конструкции действия могут быть вложенными. Области действия имен подчиняются обычным правилам блочности: имена, объявленные во вложенной конструкции, скрывают такие же имена из объемлющих конструкций. Так, в следующем примере

**do**  
 x: Integer  
 ... // 1  
 **do**  
 x: Real  
 ... // 2  
 **end**  
  
 **do**  
 y: Integer **is** x // 3  
 **end**  
 ... // 4  
 **end**

объявление объекта x во внешнем блоке действует в пределах этого блока и во вложенных блоках (фрагменты текста, обозначенные 1, 3 и 4) за исключением области действия вложенного блока (фрагмент 2), в котором объявлен объект с тем же именем. Обратите внимание, что во втором вложенном блоке (фрагмент 3) объект x также доступен, так как в этом блоке нет одноименного объявления с таким же именем.

**Блоки с пред- и постусловиями**

Конструкция, представленная в виде **do** ... **end**, - самый простой вариант конструкции действия. Для блока можно задать ряд дополнительных атрибутов, существенно расширяющих его семантику и позволяющих выразить разнообразные потребности программистов.

Прежде всего, для блока можно задать *пред*- и *постусловия*, которые образуют так называемый *верификационный контекст блока*. Иными словами, можно определить условия, при выполнении которых может начаться выполнение действий, заданных в блоке, а также условия, которым состояние программы должно удовлетворять при завершении этих действий.

Пред- и постусловия представляют собой предикаты (выражения булевского типа).

Пример блока с предусловием:

**require** N>0 **do**  
 *Объявления и операторы*  
 **end**

Семантика этой конструкции следующая. Перед началом выполнения объявлений и операторов тела блока производится вычисление выражения, заданного после служебного слова **require**. Заметим, что это вычисление может производиться как на этапе компиляции, так и непосредственно в процессе выполнения программы. Если значение выражения есть true, то начинается выполнение тела блока. В противном случае (а) если вычисление производится на этапе компиляции программы, то компилятор трактует данную конструкцию как ошибочную и выдает соответствующее диагностическое сообщение, (б) если вычисление производится в процессе выполнения программы, то объявления и операторы блока не выполняется, а возбуждается исключительная ситуация типа PreconditionViolation.

Заметим, что в приведенном примере объект N, участвующий в формуле предусловия, принадлежит некоторому внешнему по отношению к блоку контексту.

Аналогично предусловиям, можно задать проверку состояния программы в точке завершения выполнения блока. Вот как может выглядеть конструкция постусловия:

**do**  
 x: Integer  
 *Прочие объявления и операторы*  
 **ensure**  
 N>0 && x=1  
 **end**

**Блоки в составе составных операторов**

Как уже говорилось, блоки могут быть вложенными. Это правило распространяется и на другие составные операторы – условные и циклы. Иными словами, тела условных и циклов по определению представляют собой блоки. Однако в этих случаях синтаксис блоков несколько упрощается: завершающее служебное слово **end** не является необходимым, так как объемлющая составная конструкция уже включает завершающее служебное слово:

**if** *условие* **do**  
 *Объявления и операторы*  
 **elsif** *условие* **do**  
 *Объявления и операторы*  
 **else**  
 *Объявления и операторы*  
 **end**

Здесь в блоке, заданном после *условия*, нет необходимости задавать отдельный **end**, так как уже имеется завершающее служебное слово **elsif** или **else**, которое однозначно фиксирует завершение блока. Более того, в случае части **else** условного оператора начальное служебное слово **do** также не нужно. Можно сказать, что в данном случае от блока в данном случае остается только его «тело».

В случае, когда для блока в составе условного необходимо задать пред- или постусловия, его синтаксис также представляется в неполном («облегченном»), но вполне логичном и интуитивно понятном виде:

**if** *условие* **do**  
 *Объявления и операторы*  
 **ensure** *Постусловие*  
 **else**  
 **require** *Предусловие* **do**  
 *Объявления и операторы*  
 **end**

Предложенная конструкция несколько отличается от традиционного синтаксиса условных операторов, в котором после условия задается служебное слово **then**. Заметим, однако, что это слово используется далеко не во всех языках; кроме того, представляется, что **do** как альтернатива **then** обладает не меньшей выразительностью и, кроме того, обеспечивает более полную унификацию синтаксических правил.

Аналогично формируются тела циклов, например:

**while** i **in** 1..10 **do**  
 Объявления и операторы  
 **end**

Заметим, что в случае циклов более предпочтительным было бы вместо **do** использовать более выразительное служебное слово **loop**, которое повышает общую читабельность конструкции. Кроме того, предложенный синтаксис не дает возможность явно задать бесконечный цикл вида **loop** ... **end**. Однако выбор был сделан в пользу большего единообразия.

**Блоки и исключительные ситуации**

Продолжаем обсуждение конструкции «блок». Помимо пред- и постусловий, данная конструкция может быть нагружена и другой полезной семантикой. Речь идет, в частности, о перехвате и обработке исключительных ситуаций (exceptions).

Если необходимо задать реакции на исключительные ситуации, которые могут возникнуть в данном блоке (или в блоках, динамически вложенных в данный), то такая реакция представляется следующей конструкцией:

**do**  
 *Объявления и операторы*  
 **when** *Тип ситуации* **do** *Действия*  
 **when** *Тип ситуации* **do** *Действия*  
 **end**

Заметим, что внутри **when**-фраз также используется синтаксис блока, но, как и в примере с условным оператором, в «облегченном» виде, так как заключительное служебное слово **end** излишне ввиду наличия либо последующей фразы **when**, либо **end** объемлющей конструкции.

Фразы **when** могут содержать и более сложные конструкции. Так, в случае, когда обработка нескольких различных ситуаций должна быть выполнена с помощью одной и той же последовательности действий, допускается задание нескольких ситуаций в одной фразе **when**, например:

**do**  
 a **is** Integer.MaxInteger + 1  
 b **is** Integer.MinInteger - 1  
 **when** Overflow | Underflow **do** put("Abnormal termination")  
 **end**

Разумеется, блок может содержать одновременно все конструкции, введенные выше – пред- и постусловия и обработчики исключительных ситуаций. Вот пример:

**require** N>0 **do**  
 x: Integer  
 *Прочие объявления и операторы*  
 **ensure**  
 N>0 && x=1  
 **when** PreconditionViolation **do** *Actions1*  
 **when** PostconditionViolation **do** *Actions2*  
 **else** *Actions3*  
 **end**

Порядок следования постусловий и реакций на исключительные ситуации в блоке именно такой, как показано в предыдущем примере: сначала должны задаваться постусловия (если они необходимы), после них – последовательность реакций на исключительные ситуации.

Одно из объяснений жесткого порядка следования заключается в том, что семантика пред- и постусловий подразумевает возбуждение исключительных ситуаций предопределенных типов – и эти ситуации могут быть обработаны блоками из фраз **when** того же блока, которые в этом случае и должны располагаться текстуально после постусловий.

**Удаленная активация блоков**

Следующий логический шаг в рассмотрении семантики блоков заключается в возможности их *удаленной активации*. Обычно выполнение объявлений и операторов из тела блока производится, когда поток управления попадает в него. Однако возможна и более гибкая и мощная модель.

Блок можно рассматривать не просто как совокупность действий, подлежащих немедленному выполнению при попадании в них потока управления, но как действия, которые активируются в некоторый произвольный момент времени, не связанный с местом их объявления. Вот простой пример:

() **do** **return** Integer.MaxInteger-1 **end**

Этот (несколько искусственный) пример задает последовательность действий, вычисляющих некоторое значение. Подобные конструкции называются *функциональными объектами*. Существенным аспектом служит возможность обращаться с данной конструкцией как с обычным объектом – то есть, инициализировать им переменные, передавать его в качестве аргумента в подпрограммы, а также активировать заданные в них действия в произвольных точках программы, например:

// Объявление функционального объекта  
 x **is** () **do** **return** Integer.MaxInteger-1 **end** ... // Вызов функционального объекта  
 x()

Следующие примеры показывают передачу функционального объекта в качестве аргумента функции:

Allocate(x)  
 Allocate(() **do** **return** Integer.MaxInteger-1 **end**)

Как видно из последнего примера, функциональные объекты могут выступать как в качестве непосредственных аргументов подпрограмм, так и служить значениями переменных, которые передаются в качестве аргументов.

Подобные возможности поддерживаются во многих языках программирования и носят различные названия: анонимные подпрограммы, функции-литералы и т.п. Семантически схожая, хотя и более низкоуровневая и небезопасная конструкция имеется и в языке Си («указатель на функцию»).

**Параметризованные блоки**

В предыдущих примерах пустые круглые скобки можно было трактовать как своего рода маркер, позволяющий различать обычные блоки и функциональные объекты. Однако, скобки несут более глубокую семантику. В общем случае они могут содержать последовательность объявлений, смысл которых заключается в задании *параметров* блока. Иными словами, при дистанционной активации функционального объекта (или попросту говоря, при его вызове) можно динамически задать аргументы такого вызова, тем самым, делая функциональный объект более универсальным и расширяя возможные способы его применения:

a **is** (x:Integer) **do** **return** x+1 **end**  // значение a – функциональный объект  
  
 b **is** a(776) // параметризованная активация (вызов)  
 // функционального объекта  
  
 c **is** a // значением c становится функциональный объект  
 // из переменной a

В данном примере значением переменной a становится функциональный объект с целочисленным параметром. Этот объект далее активируется (вызывается) с передачей ему объекта целочисленного типа в качестве аргумента. Результат вызова определяется операторами тела блока, и этот результат задается в качестве начального значения переменной b.

Последняя строка примера показывает возможность обычного копирования функциональных объектов.

Помимо параметров, для функционального объекта можно явно специфицировать тип значения, возвращаемого этим функциональным объектом, например:

(x, y: Integer): Real **do return** sqrt(sqr(x)+sqr(y)) **end**

Во многих случаях необходимости в явном задании возвращаемого типа нет, так как компилятор может автоматически вывести возвращаемый тип из анализа тела блока (например, из выражения оператора **return**). Однако в общем случае (а также если возвращаемый тип отличается от того, который компилятор может вывести автоматически, см. пример Fact ниже) такая спецификация возможна и/или необходима.

Может возникнуть вопрос: каков тип самого функционального объекта? Подчеркнем, что речь идет не о типе значения, возвращаемого таким объектом, а о типе функционального объекта как такового.

По определению, функциональный объект имеет тип Procedure для подпрограмм, не возвращающих значение, и тип Function для случая подпрограмм-функций. Указанные типы задаются как обобщенные (generic); их определение содержится в стандартной библиотеке языка SLang. Более подробное описание этих типов приводится в главе XXX.

Следует заметить, однако, что непосредственное использование типов Procedure и Function в большинстве случаев не является необходимым. Так, во всех предыдущих примерах функциональные объекты вводились без указания их типов, так как компилятор в состоянии вывести их автоматически.

**Обобщенные функциональные объекты**

В языке SLang поддерживается механизм обобщения (типовой и константной параметризации) программных сущностей – контейнеров и подпрограмм. Представляется естественным распространить этот механизм и на функциональные объекты. Так, алгоритм вычисления максимального значения для произвольных типов может быть представлен следующей конструкцией:

[T](x, y: T) **do return if** x>y **do** x **else** y **end**

Здесь в квадратных скоках задается типовой параметр, который используется для объявления (обычных) параметров функционального объекта. Если такой функциональный объект станет значением некоторой переменной, то его активация может быть выполнена стандартным образом:

**unit** X **is**  
 >(p: **as** **this**): Boolean **do**  
 // Реализация операции сравнения  
 **end**  
 ...  
 **end**  
  
 max **is** [T](x, y: T) **do return if** x>y **do** x **else** y **end** a, b : X  
 res1 **is** max(a,b) // 1  
 ...  
 m, n: Integer  
 res2 **is** max(m,n) // 2

Как видно, активация (вызов) обобщенного функционального объекта синтаксически не отличается от активации обычного функционального объекта. При этом семантика вызова несколько отличается. Анализируя вызов max, компилятор распознает тип этой переменной как тип обобщенного функционального объекта и проведет инстанциацию (настройку) этого объекта на конкретный тип. Этот тип будет выведен из типов аргументов вызова (в случае первого вызова это тип X). На основе этой информации компилятор сгенерирует обычный (необобщенный) функциональный объект, в котором все вхождения абстрактного типа T будут заменены на конкретный тип X. Полученный объект и будет использован при вызове.

Такие же действия будут выполнены в случае второго вызова max. Здесь в качестве конкретного типа компилятор выведет тип Integer.

В некоторых случаях, однако, компилятор не может самостоятельно вывести обобщенные параметры – например, когда задаются нетиповые параметры (константы). В этом случае аргументы настройки приходится задавать явно, например:

power **is** [N: Integer](x: Integer): Long **do** res **is** 1  
 **while** i **in** 1..N **do**  
 res \*= x  
 **end  
 return** res  
 **end**  
  
 power[10](5)

**Именованные функциональные объекты**

До сих пор функциональные объекты играли роль обычных значений (констант). С этой точки зрения, функциональный объект вида (x:Integer) **do** **return** x+1 **end** является таким же значением, что и литерал какого-либо типа, например, 777 или "abc". Можно сказать, что такие объекты не имеют имен и представляют сами себя.

Следующим логическим шагом служит придание функциональному объекту *имени*. Именованный функциональный объект образует традиционное *объявление подпрограммы*:

Abs(x: Real): Real **do** **if** x<0 **do** –x **else** x **end**  
 Max[T](x, y: T): T **do if** x>y **do** x **else** y **end**

Должно быть понятно, что имена из объявлений подпрограмм и соответствующие функциональные блоки образуют постоянную (неразрывную) связь. В этом отношении объявление подпрограммы аналогично объявлению константного объекта функционального типа. Так, первое объявление из примера выше считается семантически эквивалентным такому объявлению:

**const** Abs **is** (x: Real): Real **do** **if** x<0 **do** –x **else** x **end**

Объявления подпрограмм подчиняются обычным правилам, характерным для других языков. В частности, допускается вложенность подпрограмм, а также рекурсивные вызовы:

Fact(x: Integer): Long **do  
 return if** x<=1 **do** 1 **else** x\*Fact(x-1)  
 **end**

В принципе, семантическая эквивалентность обычных объявлений подпрограмм и объявлений константных объектов функционального типа свойственна также и для неконстантных объектов. Иными словами, можно было бы разрешить и такую форму рекурсивных вызовов:

fact **is** (x: Integer): Long **do  
 return if** x<=1 **do** 1 **else** x\*fact(x-1)  
 **end**

Здесь есть, однако, ряд проблем. Поскольку в этом примере fact – переменная, то она может динамически может принимать различные значения, в данном случае – значение любого другого функционального объекта с той же самой сигнатурой. Тем самым, вызов fact в теле функционального объекта теоретически может приводить не к рекурсивному вызову того же функционального объекта с измененным аргументом, а к вызову, вообще говоря, любого другого функционального объекта. Вот пример:

unsafe **is** () **do**  
 **if** condition **do** unsafe := bad **else** unsafe := good **end**  
 unsafe()  
 ...  
 **end**

Подобные трюки вряд ли могут считаться допустимыми, и прибегать к ним не рекомендуется.

**Сокращенная форма записи блоков**

Во многих реальных случаях блоки функциональных объектов содержат весьма малый объем кода – вплоть до единственной строки. Примерами могут служить почти все фрагменты, приводимые выше.

Для упрощения и сокращения кода в подобных случаях допускается ряд сокращений. Во-первых, вместо полной формы оператора возврата вида **return** *Expression* можно задавать непосредственно выражение, если оно располагается в однозначной синтаксической позиции текстуально последним в теле блока или в теле некоторого составного оператора.

Примером может служить более компактный вариант записи функции Fact:

Fact(x: Integer): Long **do if** x<=1 **do** 1 **else** x\*Fact(x-1) **end**

Во-вторых, если тело блока представляет собой единственный оператор, то блок в целом вместо использования служебных слов **do** и **end** может быть оформлен в менее громоздком и более наглядном виде:

Fact(x: Integer): Long => **if** x<=1 **do** 1 **else** x\*Fact(x-1)

Наконец, в случаях, когда возвращаемый тип может быть выведен компилятором, его, как уже говорилось, можно опускать.

Заметим, что опускать тип возвращаемого значения имеет смысл не всегда – даже если компилятор может вывести тип автоматически. Так, для случая функции Fact компилятор, анализируя тело функции, не сможет «догадаться», что типом возвращаемого значения этой функции должен служить Long, а не Integer.

Тем не менее, во многих случаях тип возвращаемого значения выводится корректно. Вот пример более компактной записи функции Max:

Max[T](x, y: T) => **if** x>y **do** x **else** y

**Функциональные объекты и замыкания**

<Не дописано>