Язык программирования SLang

Неформальное описание

1. Введение

Предпосылки

Язык программирования, со своей инфраструктурой – системами поддержки времени выполнения, стандартными и прикладными библиотеками, инструментами и средами разработки (компиляторы, редакторы связей, IDE) представляет собой ключевой инструмент создания программного обеспечения и, тем самым, служит определяющим фактором обеспечения эффективного, безопасного и надежного функционирования многочисленных и разнообразных электронных устройств в современном мире.

Существующее положение в сфере языков программирования и, шире, в области инструментария современного программирования весьма далеко от идеала. Большинство языков программирования, наиболее широко используемых в настоящее время, создано более двадцати лет назад и в настоящее время совершенно не адекватно практически ни одному из современных требований, предъявляемых к инструментам разработки современного ПО. Эти языки архаичны, неуклюжи, громоздки, неудобны и сложны в практическом использовании, не способствуют надежности и эффективности создаваемого ПО, зачастую несут явный отпечаток вкусовых пристрастий и причудливых взглядов их создателей.

Новые языки программирования, в изобилии появляющиеся в последние пять-семь лет, пытаются преодолеть указанные недостатки, однако в значительной степени повторяют устаревшие и порочные подходы проектирования, берущие своё начало в восьмидесятых годах прошлого столетия.

Существо проекта

Основной смысл предлагаемого проекта заключается в том, чтобы спроектировать и реализовать оригинальный язык программирования (предварительное название – **SLang**) вместе с необходимой экосистемой: компилятором, подсистемой поддержкой времени выполнения, стандартными библиотеками, редактором связей (комплексатором).

Основной смысл проекта заключается в том, чтобы предложить разработчикам программного обеспечения XXI века инструмент, который бы позволил им решать разнообразные задачи наиболее простым и надежным способом в соответствии с предъявляемым требованиям и применительно к разнообразным уровням квалификации разработчиков.

Язык SLang воплощает адекватное понимание существа процесса проектирования и разработки современного ПО. Он включает свойства, обеспечивающие надежность, безопасность и эффективность программ, создаваемых с его использованием. В то же время он достаточно прост для обучения, освоения и использования, что обеспечит «гладкий» процесс разработки и сопровождения, а также предоставит возможность включить в сферу разработки ПО более широкие, нежели в настоящее время, сообщества разработчиков, в том числе, непрофессионалов.

Язык SLang носит принципиально универсальный характер, и потому будет пригоден для успешного создания ПО в различных сферах применений – от микроустройств с минимальными характеристиками производительности и памяти

(что особенно существенно в сфере «интернета вещей», IoT), мобильных устройств, приложений для сети Интернет, до серверных систем, суперкомпьютеров и систем реального времени.

Язык SLang – масштабируемый; под этим подразумевается его пригодность для реализации программных систем различного масштаба и сложности: от многофункциональных, логически насыщенных систем с повышенными требованиями к надежности, до коротких программ («скриптов»), решающих простые прикладные задачи.

Помимо языка, проект будет включать необходимый (на первом этапе – минимальный, в итоге – исчерпывающий) набор инструментов, в совокупности обеспечивающих полный жизненный цикл программ.

Язык спроектирован как машинно-независимый. Его семантика имеет высокий уровень и не ориентирована на особенности какой-либо аппаратной или программной платформы. В то же время, язык может быть эффективно реализован для любой распространённой в настоящее время среды. К ним относятся Linux и её наиболее популярные клоны, версии и сборки, iOS (Apple), Windows и Windows Phone, а также специализированные среды, в том числе, встроенные системы.

Замечание об импортозамещении

В настоящее время достаточно остро стоит вопрос создания отечественных версий программного обеспечения различного рода. Речь идет о полном спектре программных средств – от программ прикладного характера до базового, системного ПО – операционных систем, систем управления базами данных (СУБД), встроенных систем специального назначения.

Не вдаваясь в обсуждение нетехнических аспектов этой тенденции, следует заметить, что общей проблемой, связанной с информационной независимостью государства, является вопрос *инструментальных средств*, используемых для построения отечественного ПО. В самом деле, программная система, полностью разработанная силами отечественных специалистов, чтобы превратиться в работающую программу, должна быть преобразована в машинное представление, подлежащее выполнению на некотором аппаратном оборудовании. Такое преобразование производится посредством специальных *инструментов* – компиляторов языков программирования, компоновщиков (редакторов связей), статических анализаторов программ и т.д. Представляется очевидным, что полная информационная независимость может быть достигнута только в случае создания и применения отечественных инструментальных средств.

К сожалению, этот, на наш взгляд, ключевой аспект зачастую оставляется без внимания. По нашему мнению, усилия, предпринимаемые в направлении создания отечественного ПО, должны в обязательном порядке подразумевать разработку отечественных инструментальных средств. Проект языка программирования SLang и системы программирования на его основе призван заполнить существующий пробел и предложить мощное и современное решение в указанной сфере.

2. Краткая характеристика языка SLang

Модульность

В отличие от многих современных языков, программа на SLang формируется по модульному принципу: строительными блоками любой программы служат *модули* – независимо определяемые, независимо хранимые и независимо компилируемые единицы со строго определенными интерфейсами, согласно которым они могут вступать в различные отношения друг с другом: использование, агрегация, наследование и т.д.

В языке определены два основных вида модулей: контейнеры и подпрограммы. Контейнеры представляют агрегацию логически связанных ресурсов (данных и подпрограмм-членов), подпрограммы реализуют некоторую функциональность и, в свою очередь, могут представлять собой процедуры или функции.

Продолжая эту логику, естественно считать, что компонент любого из указанных видов может служить *единицей компиляции*, то есть, допускать раздельную компиляцию. Отсутствие ограничений способствует созданию композиции программы, адекватной требованиям и особенностям ее использования. Программа может представлять собой, по сути, произвольную композицию единиц, от простого набора взаимодействующих подпрограмм до сложной комбинации контейнеров различных видов.

В предельном случае единица компиляции или вся программа может представлять собой единственную единицу – простую последовательность операторов. Если необходимо написать несколько строк кода, которые будут служить реакцией на нажатие клавиши мыши в некотором средстве просмотра Интернет-страниц, то нет необходимости писать подпрограмму – достаточно просто задать последовательность операторов, которая выполнит нужное действие. Если же решение задачи предполагает более сложную логику, то результат можно получить, комбинируя отдельные подпрограммы, контейнеры и блоки. Таким образом, единая языковая нотация может быть использована для решения максимально широкого класса задач.

Строгая типизация

Понятие типа является одним из базовых понятий любого языка. Под ***типом*** некоторого объекта, существующего в программе, понимается тройственная сущность, определяемая множеством значений, которые может принимать данный объект, связанным с ним множеством ***операций***, допустимых над значениями данного типа, а также множеством ***отношений*** между данным типом и другими типами.

Язык SLang представляет собой язык со ***статической типизацией***. Это означает, что тип является статически неизменным свойством объекта; это свойство присуще объекту с момента его возникновения в программе и не может измениться во время жизни этого объекта. В терминах программирования это означает, что тип объекта назначается объекту (явно или неявно) при его объявлении, и не существует возможностей (языковых конструкций), позволяющих изменить тип в процессе выполнения программы.

> Статическая типизация является в настоящее время признанным средством обеспечения надежности и высокой производительности программ.

Тип может быть (явно) приписан объекту программистом при объявлении объекта, либо (неявно) выведен компилятором из контекста объявления этого объекта.

Примером контекста в данном случае может служить тип инициализирующего выражения из объявления объекта.

> Оба варианта назначения типа – явный и неявный – являются статическими по своей природе и отличаются друг от друга только агентом, производящим назначение типа объекту: в первом случае таким агентом является программист, во втором – компилятор.

Текущее значение объекта может быть преобразовано в значение другого типа. Правила преобразования типов определяются свойствами этих типов, представленными в форме соответствующих операций; предопределённые правила преобразования в языке отсутствуют.

Поддержка различных парадигм программирования

SLang – мультипарадигменный язык, в том смысле, что в нём воплощены важнейшие современные концепции программирования, включая объектно-ориентированное, обобщённое (generic) и функциональное программирование.

В языке имеется понятие типа (класса), которое реализуется посредством языковой конструкции «контейнер» (см. ниже), со всеми традиционными свойствами – инкапсуляцией имён, наследованием, полиморфизмом. Имеется возможность задавать абстрактные классы, а также реализовывать между классами отношения множественного наследования.

Любой контейнер или подпрограмма может быть параметризована типом (типами), либо константными значениями. Настройка обобщенных программных единиц предполагает задание конкретных типов и/или значений. Тем самым, в языке обеспечивается полная поддержка парадигмы обобщенного программирования, что позволяет проектировать компоненты программы в виде, максимально независимом от контекстов их использования.

> Средства обобщённого программирование, реализованные в языке, представляют определённый компромисс между полным, но весьма громоздким и трудным для использования механизмом шаблонов языка C++ и «облегчёнными», но явно недостаточными для практического программирования средствами типовой параметризации Java, C# и Eiffel.

В языке поддерживается необходимый набор средств функционального подхода к программированию, включая лямбда-выражения, замыкания. Эти свойства основаны на трактовке функций как значений, а также предполагают свободное использование понятия неизменяемых (immutable) объектов.

> Реализация элементов функционального программирования в языке SLang носит в большой степени «инженерный» характер, схожий по ассортименту средств с аналогичными средствами C++ и C#. Поддерживается только минимально необходимый «функциональный» набор. Реализация функциональной парадигмы в полном объёме, свойственная таким языкам, как Haskell или F#, по мнению авторов, потребует радикальную перестройку мышления разработчиков, что может усложнить изучение языка и отпугнёт разработчиков излишней и сложностью.

Контейнер: модуль, класс и тип в одном флаконе

Важнейшими концепциями, используемыми при разработке программного обеспечения (ПО), служат понятия атрибутов (данных) и подпрограмм (действий). Атрибуты 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.

Более строго, ***контейнер*** (***unit***) можно определить как поименованную совокупность атрибутов и подпрограмм, которая может быть параметризована либо типом, либо константой, 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.

Контейнер можно рассматривать как определение множества данных и операций над ними – то есть, как задание некоторого **типа**. Тем самым, можно определить объект, тип которого будет 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 C++ 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 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. Некоторые авторы приводят 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 и Oberon) выглядит несколько искусственным – при многих других несомненных достоинствах этих языков.

В языке SLang сохраняются преимущества единой нотации задания контейнеров, с возможностью явного задания *различных способов использования* контейнеров.

Однородная система типов

Система типов языка SLang является *однородной*. Это означает, что в языке отсутствует деление на различные категории типов (например, «встроенные в язык» и «определяемые пользователем»). Любой тип, используемый в программе, определяется единообразно, посредством универсальной конструкции

«контейнер» (unit). Единственное различие в системе типов заключается в том, что некоторые наиболее часто используемые на практике типы – целый, вещественный, булевский, а также такие структуры, как массивы, списки, словари и т.д. – определены в качестве библиотечных, и для операций над объектами таких типов компилятор может порождать более эффективный объектный код. Типы, определяемые пользователем, используются наравне с библиотечными типами без каких-либо ограничений.

Понятие контейнера предоставляет удобное и универсальное средство создания разработчиком новых типов на основе существующих. При этом допускаются все традиционные методы и практики определения типов, в том числе агрегирование и наследование.

Возможность создавать и использовать в программе некоторый новый тип – необходимое условие гибкости языка и, как следствие, залог его возможностей по адекватной реализации множества реальных структур данных и отношений между ними. Использование существующих типов в качестве составных частей других типов в значительной степени обеспечивает мощность языка и выразительность программ, написанных на этом языке.

Однородность системы типов существенно упрощает понятийный базис языка, делая его стройным, логичным, простым для понимания, тем самым обеспечивая максимально возможную простоту, ясность, недвусмысленность и надёжность программ. В то же время, однородность не ограничивает выразительные возможности языка, предоставляя в распоряжение разработчика полный спектр инструментов для создания структур данных любой сложности.

Объектный подход и множественное наследование

Как уже говорилось, язык в полном объёме поддерживает весь сложившийся к настоящему времени комплекс средств объектно-ориентированного программирования: понятие класса (выраженное в конструкции «контейнер»), наследование с возможностями управления характером наследования, полиморфизм с абстрактными классами и методами.

В отличие от многих современных языков, наследование в языке SLang реализовано полностью: контейнер может наследовать свойства нескольких других контейнеров.

> Сведение понятия наследования к единичному, характерное для некоторых современных языков (Java, C#, Python) продиктовано, скорее, техническими причинами – желанием упростить реализацию механизма наследования, избавиться от возможных неоднозначностей и сделать программы более понятными для читателя. В то же время, такое ограничение существенно сужает возможности разработчиков по адекватной реализации требований к архитектуре программных комплексов.

Семантика наследования в SLang определена таким образом, чтобы избавить программиста от необходимости обеспечивать безусловную корректность полного графа наследования. Контроль корректности привязан к точкам использования сущностей из базовых классов.

Контрактное программирование

Язык SLang поддерживает полный спектр механизмов *контрактного программирования*, включая пред- и постусловия для подпрограмм и инварианты контейнеров и циклов. Система поддержки времени выполнения обеспечивает эффективную (параллельную, где это возможно) проверку условий и инвариантов.

> Подход к проектированию программ на основе понятия *контракта* (Design by contract ©), изначально разработанный и обоснованный Б.Майером и реализованный в языке Эйфель, в настоящее время является общепринятым средством повышения надежности и верифицируемости ПО. Подход в том или ином объеме реализован во многих современных ЯП.

Параллельное программирование

В отличие от большинства современных языков, где поддержка параллельности реализована на уровне библиотек и носит ограниченный и слабо верифицируемый характер, SLang включает удобный и достаточно надежный механизм распараллеливания программ на уровне самого языка. В языке имеется простой и компактный набор конструкций и спецификаторов для задания многопоточности и синхронизации по доступу к данным. Кроме того, семантика языка допускает автоматическое распараллеливание исполнения.

Безопасность

Проблема, связанная с неконтролируемым использованием нулевых указателей («пустых» или «повисших» ссылок), является одной из наиболее распространенных в практике программирования, а также одной из самых опасных по своим последствиям с точки зрения обеспечения надежности программ. В то же время, контроль доступа по таким указателям не имеет удовлетворительного решения в традиционных языках.

В языке SLang проблема пустых указателей трактуется не как самостоятельная проблема, а как часть более общей проблемы некорректной работы с *неинициализированными атрибутами*. Пустая ссылка считается разновидностью неинициализированного атрибута, и в языке имеются механизмы, которые строго ограничивают случаи, когда нам действительно нужны неинициализированные атрибуты, от ситуаций, когда всякая сущность должна иметь определенное значение. В дополнение к этому имеется надежный механизм перехода от потенциально неинициализированных атрибутов к инициализированным – своего рода мостик от «опасного» мира в «безопасный».

Замечание о синтаксисе

Обычно, говоря о каком-либо языке программирования, в первую очередь имеют в виду правила составления его конструкций – то есть, синтаксис. Под выражением «выучить язык», как правило, понимают освоение именно этих правил. Более того: многие руководства по новым ЯП в первых же строках как бы успокаивают читателя: мол, если вы знаете язык X, то вам не составит труда выучить и наш язык, так как мы сделали его *очень похожим* на ваш любимый язык X! И в подавляющем большинстве случаев под «похожестью» имеется в виду именно синтаксис.

Понятно, что подобного рода сентенции имеют отчетливый маркетинговый характер, имея своей целью привлечь на свою сторону программистов, интересующихся новинками, но опасающихся чрезмерных усилий и времени на их освоение.

Но это еще полбеды; гораздо важнее (и опаснее!), что такие успокоительные заверения маскируют существенные смысловые, *семантические различия* между языками. В самом деле: если бы новый язык не несёт в себе каких-то сущностных нововведений, новых понятий, позволяющих решать новые сложные задачи – зачем было придумывать его?

> Хорошим примером служит обманчивая «похожесть» языков C++ и C#/Java, которую можно проиллюстрировать на следующем разительном примере.

Пусть в программах на C++, C# и Java имеются объявления классов вида `class C { ... }`. Эти объявления внешне будут выглядеть в значительной степени одинаково (синтаксическая различия небольшие). И пусть во всех трёх программах встречается объявление вида `C c;`. Синтаксис такого объявления одинаков во всех языках, но какова семантика такого объявления?

В языке C++ объявление объекта классового типа имеет существенный смысл и подразумевает выделение памяти (в данном случае – в стеке) для этого объекта и вызов *конструктора* умолчания класса C, который собственно, и создаёт объект данного класса.

В то же время *точно такое же* объявление в языках C# и Java имеет *совершенно другой смысл* и заключается просто в образовании ссылки на (несуществующий) объект класса C и присваивании этой ссылке нулевого значения `null`.

Полная семантическая аналогия упомянутых объявлений в этих двух языках имеет следующий вид:

```
C c = new C();      // Java
C* pc = new C();    // C++
```

Этот пример (а подобные ложные аналогии можно множить) показывает истинную цену успокоительным заявлениям о схожести различных языков.

При проектировании языка SLang авторы не ставили задачу следовать повсеместно принятым в настоящее время синтаксическим и лексическим традициям и тем самым «понравиться» программистам. Если некоторое понятие, существенное для языка или предметной области, требует введения новой и непривычной нотации, такая нотация вводится в язык. То же относится к «стилю синтаксиса» – распространённым в настоящее время традициям обозначений блоков (областей действия), служебным словам, разделителям и т.д. Основным мотивом выбора лексики или синтаксиса для той или иной конструкции языка служит её наглядность, ясность для понимания и удобство чтения. Мы следуем парадигме, впервые высказанной создателями языка Ада, согласно которой программы в той же степени являются средством общения между людьми, как и средством общения людей с компьютерами.

В то же время в следовании таким установкам авторы не собираются становиться фанатиками и не хотят доходить до абсурда. Конструкции языка не должны быть излишне громоздкими и многословными, удивлять разработчиков и читателей программ своей экзотичностью и заставлять их прикладывать чрезмерные усилия для их освоения и понимания. По этой же причине в языке отсутствуют средства его произвольного изменения, такие, как макросы, синтаксические «расширители» и средства задания собственных операций.

3. Способ синтаксического описания

Формальная структура конструкций языка задаётся в данном документе посредством совокупности правил, записанных в модифицированной нотации БНФ (форма Бэкуса-Наура).

Каждое правило имеет следующий вид:

```
Синтаксическое-понятие
      : Последовательность-терминалов-нетерминалов-и-метасимволов
```

Здесь слева от метасимвола `:` записывается имя определяемого понятия, в справа от этого символа задаётся последовательность, определяющая структуру понятия в терминах терминальных и нетерминальных символов.

Терминальный символ (терминал) – это конечный, атомарный элемент языка, который не определяется в терминах других символов, а используется в точности в том виде, в каком он указан в правиле. Примерами терминалов могут служить разделители, знаки операций, служебные слова языка и т.д. Внутренняя структура терминалов (которые также называются лексическими элементами, или лексемами) языка описывалась в предыдущих разделах документа и далее, для целей синтаксического описания, считается несущественной. В правилах нетерминалы записываются моноширинным шрифтом синего цвета, например, `val` или `]`.

Нетерминальные символы обозначают составные конструкции языка. В совокупности, множество нетерминалов задает синтаксис программ на языке SLang. Нетерминальные символы записываются моноширинным шрифтом лилового цвета, например, `Определение-типа`.

Простейшей формой правила является последовательное задание терминалов и нетерминалов в его правой части. Такая форма означает, что определяемое в левой части синтаксическое понятие эквивалентно последовательности конструкций, заданных в правой части. Для задания более сложных правил построения конструкций служат метасимволы.

Нотация предусматривает использование следующих метасимволов:

- `|` (вертикальная черта)
  Символ служит для задания двух или более альтернатив, каждая из которых является допустимой.
- `[ ]` (квадратные скобки)
  Символы, заключенные в квадратные скобки, считаются опциональными (необязательными). Это означает, что определяемая синтаксическая конструкция считается допустимой как при вхождении данных символов, так и в случае их отсутствия.
- `{ }` (фигурные скобки)
  Символы, заключенные в фигурные скобки, могут повторяться в пределах определяемой конструкции ноль или более раз.
- `( )` (круглые скобки)
  Круглые скобки используются для простого группирования других конструкций, когда они входят в качестве составных частей в другие конструкции.

В качестве примера рассмотрим следующее правило:

```
Список-элементов
     : Элемент { , Элемент }
```

Данное правило представляет собой традиционную форму задания повторяющихся элементов языка и обозначает следующее: конструкция, называемая Список-элементов, обозначает либо единственную конструкцию Элемент (правило для которой вводится отдельно), либо произвольное число вхождений этих конструкций, отделяемых друг от друга терминальным символом «запятая».

Примером более сложного правила может служить следующее:

```
Тип : [ ref | val ] Определение-типа [ Уточнение-типа ]
```

Это правило вводит новое понятие, обозначаемое нетерминальным символом Тип. Это понятие определяется в виде последовательности трёх составных частей. Первая часть представляет собой необязательную (заключённую в квадратные скобки) конструкцию, содержащую две альтернативы, каждая из которых является нетерминальным символом. Это означает, что конструкция Тип может начинаться с вопросительного знака, либо со служебного слова **ref** или **value**, или вообще не содержать ни одного из этих символов.

Далее идет конструкция, обозначенная нетерминалом Определение-типа. Правило для этого нетерминала задаётся отдельно. После него в составе конструкции Тип может присутствовать необязательная конструкция, определяемая нетерминалом Уточнение-типа.

> Следует иметь в виду, что речь идёт о формальной структуре программ; некоторые смысловые особенности, правила и ограничения не могут быть выражены посредством формальной синтаксической нотации и потому вводятся в словесной форме в тексте настоящего документа. Так, в приведённом выше примере правила не указано, какие конкретно варианты Уточнения-типа могут следовать после Определения-типа, а для каких типов уточнения не допускаются. Обстоятельства подобного рода описываются неформально в комментариях к синтаксическим правилам.
>
> Кроме того, грамматические правила, описывающие синтаксическую структуру языка, в совокупности не являются полными и непротиворечивыми. Как и в определениях многих других языков, данные правила служат информационным целям, представляя общие правила формирования программных конструкций и программ в целом.

4. Лексическая структура языка

Текст программы

Текст программы представляет собой последовательность *символов* стандарта Unicode, организованную по правилам, определённым в настоящем описании. Физический размер представления символов, образующих программу, а также их кодировка определяются реализацией. В Приложении 1 содержатся минимальные требования к представлению символов программ, а также рекомендации по использованию различных систем кодировок (UTF-8, UTF-16 и т.д.).

Один или несколько последовательных символов образуют *лексемы* (*tokens*) – элементарные единицы программы. В свою очередь, из последовательностей лексем по определённым в языке *синтаксическим правилам* строятся все конструкции программы – контейнеры, подпрограммы, объявления, операторы и выражения. Синтаксические правила определяются посредством специального метаязыка (см. раздел X) и в совокупности образуют его *формальную грамматику*.

Каждая конструкция языка имеет конкретный смысл – *семантику*, которая объясняется неформально в словесной форме в соответствующем разделе.

Физические носители текстов программ

Понятие «исходного файла» или «файла с исходным текстом программы» (source file) в языке не определяется. Программная единица может быть получена из источника, природа которого может быть произвольной и не влияет на семантику этой программной единицы. Как пример, программная единица может поступать из порта ввода-вывода, загружаться извне по локальной или глобальной сети, располагаться в оперативной памяти компьютера, вводиться с консоли или храниться в виде файла на жестком диске или другом носителе информации.

Единственным требованием, предъявляемым к источнику текста программной единицы, является его (текста) синтаксическая завершённость в пределах данного источника. Иными словами, текст программной единицы, содержащийся в данном конкретном источнике, должен представлять собой синтаксически целостную конструкцию.

> Заметим, что исходная программа не обязательно должна представлять собой *текст* – то есть, последовательность байтов, организованных согласно лексическим и синтаксическим правилам входного языка. Текст – это лишь одна из возможных визуальных представлений программы. Программа может, вообще говоря, иметь форму некоторой базы данных, совокупности UML-диаграмм или какой-либо другой вид.

Тексты программных единиц на языке SLang не содержат какой-либо информации об окружении программы, включая информацию о местонахождении других единиц или самой единицы, соответствующего объектного кода, о версии программного текста и т.д. Подобная информация не относится к семантике программы, а определяет технические аспекты её обработки – компиляцию, редактирование связей и комплексацию, и может передаваться соответствующим процессорам отдельно от собственно программных текстов – например, в виде *конфигурационного файла*.

Если физическим носителем исходного текста является дисковый файл, то на имя этого файла и на его местонахождение в файловой системе не накладывается никаких требований и ограничений. Имя файла, вообще говоря, не имеет никакого

отношения к имени (именам) программных единиц, содержащихся в нём. В частности, отсутствуют какие-либо требования на расширение имени файла.

> Если по каким-либо причинам необходимо определить некоторую дисциплину именования файлов, то рекомендуется расширение `.slang`. Сопутствующие файлы, например, содержащие объектный код той или иной целевой платформы, могут иметь расширения, рекомендуемые или требуемые данной платформой.

Лексемы

Лексическая структура языка не содержит практически никаких необычных элементов, которые в том или ином виде не встречались бы в других распространённых ЯП. Элементарной единицей текста программы является ***лексема*** (***token***) – максимальная слитная (не содержащая пробельных символов) последовательность символов, имеющая в языке определённый смысл.

Лексемы могут образовываться как из единственного символа, так и состоять из двух или более подряд идущих символов. В процессе первичной обработки текста входной программы лексемой считается самая длинная последовательность символов, образующая лексему из числа определённых в языке. Например, последовательность, состоящая из символа двоеточия (`:`) и символа «равно» (`=`), всегда считается лексемой «присваивание» (`:=`) и никогда не рассматривается как последовательность односимвольных лексем «двоеточие» и «равно» (хотя такие лексемы и определены в языке).

Пробельные символы

Пробельные символы (whitespaces) не являются составными частями ни лексем (за исключением символьных и строковых литералов), ни синтаксиса конструкций языка, не несут собственной семантики и служат для отделения лексем друг от друга, а также для достижения большей наглядности и читабельности текста программы в целом. Использование пробельных символов является необязательным, кроме случаев, когда слитное написание лексем может вызвать неоднозначность в их идентификации.

К пробельным символам относятся:

- Пробел (blank), код `\u0020`.
- Символ горизонтальной табуляции (horizontal tab), код `\u0009`.
- Символ конца строки (end line), код `\u000D`.
- Символ перевода каретки (line feed), код `\u000A`.

Комментарии

Комментарии предназначены для целей неформального аннотирования текстов программ и не несут какой-либо семантической нагрузки. С лексической точки зрения комментарии считаются пробельными символами.

Правила написания комментариев в точности повторяют правила, принятые во многих современных ЯП, таких как C++, Java, C#:

- *Короткий комментарий* начинается со слитной последовательности символов “`//`” и завершается символом новой строки.
- *Длинный комментарий* – это любая последовательность символов (в том числе пробельных символов), заключённая между комбинациями «`/*`» и «`*/`», включая сами эти комбинации. Допускается вложенность длинных комментариев.

Как следует из определения, длинный комментарий может располагаться на нескольких соседних строках текста.

Помимо двух видов комментариев, в языке имеются так называемые «значащие» или *документирующие* комментарии. Документирующий комментарий начинается с трёх подряд идущих символов наклонной черты и продолжается до символа новой строки. Символы, образующие документирующий комментарий, могут включать специальные XML-теги, которые выделяют стандартные разделы комментария. В системе программирования SLang предусматривается специальный инструмент, производящий извлечение значащих комментариев из текста программы и выполнение ряда полезных операций с ними (в частности, формирование документации на программу). Подробнее об этом см. Приложение X.

Категории лексем

Набор лексем, допустимых в языке, можно разделить на следующие категории:

- Разделители
- Знаки операций
- Идентификаторы
- Служебные слова
- Литералы (явные константы, manifest constants)

Разделители

Разделители служат вспомогательным целям и предназначены для повышения наглядности программ, в частности, отделяя однородные конструкции друг от друга, а также определяя относительный порядок вычислений в выражениях.

К разделителям относятся следующие лексемы:

- Одинарная и двойная кавычки `' "`
- Запятая `,`
- Точка `.`
- Точка с запятой `;`
- Двоеточие `:`
- Круглые и квадратные скобки `( ) [ ]`
- Присваивание `:=`
- Встроенный генератор значений `..`
- Знак «стрелка вправо» `=>`
- Вопросительный знак `?`

Заметим, что разделитель `:=`, обозначающий присваивание, строго говоря, считается *знаком операции*, см. ниже.

Использование точек с запятой

Разделитель «точка с запятой» традиционно используется либо для отделения операторов и объявлений друг от друга, либо в качестве стандартного завершителя операторов и объявлений. На практике обязательное использование для этих целей точек с запятой приводит к загромождению и «утяжелению» программы. В то же время, такие разделители в большинстве случаев не являются необходимыми и тем самым представляют собой «синтаксический мусор». По этой причине язык допускает использование в качестве разделителей операторов и объявлений, наряду с точками с запятой (или вместо них), *символов новой строки* или *символов пробела*.

Заметим, что это правило не ограничивает возможности программиста организовывать текст программы согласно принципам свободного формата. В частности, выражения могут располагаться на одной или нескольких строках без каких-либо ограничений: синтаксис языка спроектирован таким образом, чтобы исключить возможные неоднозначности в трактовке выражений, заданных в нескольких подряд идущих строках.

> Это общее синтаксическое правило, выполнение которого контролируется компилятором. Конкретный стиль программирования (например, «всегда ставить точки с запятой после каждого объявления или оператора» или «располагать в строке только одно объявление или один оператор без использования точек с запятой») может определяться руководствами по стилю программирования, принятыми в той или иной организации или команде разработчиков.
>
> Следует, тем не менее, понимать, что свободное использование пробелов для отделения языковых конструкций друг от друга может снижать наглядность и читабельность программ. Например, стиль программирования, показанный в следующем фрагменте:
>
> ```
> a:= b c := d if a>b then a.foo() else d.foo() end
> ```
>
> вполне корректен с синтаксической точки зрения, однако вряд ли может считаться приемлемым и не рекомендуется.

Знаки операций

Знаки операций используются в выражениях, задаваемых в традиционной инфиксной или префиксной нотации. Знаки операций обозначают определённые действия над одним или двумя операндами.

В языке зафиксирован конечный (нерасширяемый) список знаков, которые можно использовать для задания операций. Знаки операций образуются из одного или двух символов. Если знак представляется двумя символами, то они должны следовать непосредственно друг за другом, без пробелов.

Ниже приводится полный список знаков, которые могут использоваться для обозначения операций:

Знак	Описание
+	«Плюс»
-	«Минус»
*	«Звездочка»
/	«Прямая наклонная черта»
**	«Две звездочки»
|	«Вертикальная черта»
&	«Амперсанд»
^	«Карет»
~	«Тильда»
<	«Меньше»
<=	«Меньше» и «равно»
>	«Больше»
>=	«Больше» и «равно»

=	«Равно»
/=	«Наклонная черта» и «равно»
<>	«Больше» и «меньше»
:=	«Двоеточие» и «равно»

Очень важно отметить, что семантика (смысл) операций, обозначаемых приведёнными выше знаками, в языке не определяется. Правило заключается в том, что в любом контейнере можно объявить функцию, имя которой представляется одним из приведенных знаков. Тем самым, такая функция будет представлять собой реализацию операции, обозначенной данным знаком, для объектов типа контейнера.

Подробное описание упомянутых языковых возможностей, а также информация об относительных приоритетах, ассоциативности, арности и формальных параметров операций содержится в разделе ZZZ.

Здесь только отметим, что наряду с приведёнными выше знаками, операции могут представляться также служебными словами, причём некоторые знаки и служебные слова программист может определить как синонимы («алиасы»). Такое уподобление поддерживается по историческим причинам, а также для сохранения привычной программистам нотации. Например, знак & (амперсанд) часто используется для обозначения булевской операции «и»; в то же время, в некоторых языках программирования эта операция обозначается служебным словом **and**. Посредством механизма алиасов язык SLang предоставляет возможность сделать два способа представления этой операции равноправными.

В заключение заметим, что для некоторых операций соответствующих знаков не предусмотрено; они представляются только служебными словами.

Формальное лексическое правило для знаков операций выглядит следующим образом:

```
Знак-операции:
        + | -  | * | /  | ** | & | ^ | <
          | <= | > | >= | =  | /= | <> | :=
```

Идентификаторы

Идентификаторы представляют собой наглядные обозначения сущностей языка – контейнеров, переменных, констант, типов, меток операторов, а также имен подпрограмм.

Идентификаторы представляют собой слитную последовательность символов произвольной длины, состоящую из букв, цифр, знаков подчёркивания и знаков доллара. Первым символом последовательности, образующей идентификатор, должен служить символ буквы или символ подчёркивания. Различение идентификаторов производится по всем входящим в них символам. Символы букв в нижнем и верхнем регистрах всегда считаются различными.

```
Идентификатор:
     Буква { Буква | Десятичная-цифра | _ | $ }
```

Точный смысл понятия «буква» зависит от системы кодировки символов, используемой в конкретной реализации. В частности, для случая кодировки Unicode под «буквой» понимается любой символ, попадающий в категории Ll (lower case letters - строчные буквы), Lu (upper-case letters – заглавные буквы), Lt (title case letters), Lo (other letters – прочие буквы) или Nl (letter numerals). Тем самым, допускается задание идентификаторов на любом естественном языке,

символы которого присутствуют в системе кодировки Unicode,– на греческом, корейском, русском и т.д. Если в реализации поддерживается более узкое подмножество символов, то под буквой понимается любой символ, соответствующий латинским буквам в нижнем или верхнем регистре.

Идентификаторы, включающие символ доллара, используются для внутренних нужд компилятора и инструментов автоматической генерации программ. Знак доллара не должен использоваться для задания имён в пользовательской программе; в противном случае поведение программы считается неопределённым.

Идентификаторы: замечание о стиле

В определении языка отсутствуют какие-либо дополнительные требования на способ задания идентификаторов для различных категорий сущностей языка (например, для типов, контейнеров, подпрограмм), а также не приводятся никаких рекомендаций по стилю именования сущностей. Предполагается, что любой пользователь языка – как индивидуальный, так и коллективный – может вводить и использовать собственные правила и нормы именования.

Стиль именования сущностей, принятый в настоящем тексте (имена контейнеров начинаются с заглавных букв, остальные имена – со строчных), не является обязательным, равно как и любые другие стили.

Служебные слова

В языке существует ограниченная группа идентификаторов, которые зарезервированы для использования в целях построения различных составных конструкций языка. Такие идентификаторы называются ***служебными словами*** (***keywords***). В языке имеется 42 служебных слова[1]:

```
abstract   alias      and        as         break      check
concurrent const      else       elsif      end        ensure
extend     external   final      hidden     if         in
init       invariant  is         loop       new        not
old        or         override   pure       ref        require
return     routine    safe       super      then       this
unit       use        val        variant    while      xor
```

Перечисленные служебные слова не могут использоваться в качестве идентификаторов, именующих те или иные сущности программы; в противном случае возникает ошибка компиляции.

Литералы

Литералы (***literals***, ***manifest constants***) – это явные изображения константных значений библиотечных типов, а именно – типов, заданных контейнерами `Boolean`, `Integer`, `Real`, `Char` и `String`. Лексические правила, определяющие построение литералов, в целом соответствуют общепринятым традиционным правилам.

```
Литерал
      : Целочисленный-литерал
      | Вещественный-литерал
      | Символьный-литерал
      | Строковый-литерал
```

Целочисленные литералы

---

[1] Приведённый список служебных слов неполный и не окончательный. Он приобретёт законченный вид, когда будут согласован синтаксис всех основных конструкций языка.

Целочисленные литералы задаются посредством обычной алгебраической позиционной записи с использованием цифровых символов. При задании литералов для целей большей наглядности могут использоваться символы подчёркивания, отделяющие цифры или группы цифр друг от друга. Наличие или отсутствие символов подчёркивания не влияет на значение литерала. Например, литералы `123456789`, `123_456_789` и `12_34_56_78__9` обозначают одно и то же значение.

```
Целочисленный-литерал
      : Десятичная-константа
      | 0o Восьмеричная-константа
      | 0x Шестнадцатеричная-константа
      | 0b Двоичная-константа
```

Целочисленные литералы могут представляться в десятичной, восьмеричной, двоичной или шестнадцатеричной формах. Все перечисленные формы эквивалентны в том смысле, что одно и то же целочисленное значение может быть задано в любой из этих форм.

```
Десятичная-константа
      : Десятичная-цифра { Десятичная-цифра | _ }

Восьмеричная-константа
      : Восьмеричная-цифра { Восьмеричная-цифра | _ }

Шестнадцатеричная-константа:
     Шестнадцатеричная-цифра { Шестнадцатеричная-цифра | _ }

Двоичная-константа
      : Двоичная-цифра { Двоичная-цифра | _ }

Двоичная-цифра
      : 0 | 1

Восьмеричная-цифра
      : Двоичная-цифра | 2 | 3 | 4 | 5 | 6 | 7

Десятичная-цифра
      : Восьмеричная-цифра | 8 | 9

Шестнадцатеричная-цифра
      : Десятичная-цифра
      | ( a | A ) | ( b | B ) | ( c | C ) | ( d | D )
      | ( e | E ) | ( f | F )
```

Примеры:

Десятичные константы	`12345`, `01_234_567`
Восьмеричные константы	`0o7777`
Шестнадцатеричные константы	`0xCAFE_BABE`
Двоичные константы	`0b010100011`

Целочисленные литералы считаются объектами типа `Integer` из стандартной библиотеки языка.

Предельные (максимальное и минимальное) значения, представимые с помощью целочисленных литералов, определяются реализацией. Определённые в реализации максимальные и минимальные значения для объектов целочисленного типа могут быть получены посредством доступа к константам `Max` и `Min`, соответственно, например, `Integer.Min` и `Integer.Max`.

Вещественные литералы

Вещественные константы образуются по традиционным правилам, с заданием дробной части (мантиссы) и/или порядка числа. Вариант без задания порядка называется вещественной константой в форме с фиксированной точкой; вариант с заданием дробной части и порядка называется вещественной константой в форме с плавающей точкой.

Вещественные константы могут задаваться только в десятичном виде.

```
Вещественный литерал
      : Целая-часть (e|E) [+|-] Порядок
      | Целая-часть . Мантисса [ (e|E) [+|-] Порядок ]

Целая-часть
      : Десятичная-константа

Мантисса
      : Десятичная-константа

Порядок
      : Десятичная-константа
```

В случае задания вещественного литерала с дробной частью, согласно приведённому синтаксису, слева и справа от символа «точка» необходимо обязательное наличие чисел, обозначающих целую часть и мантиссу вещественного литерала. Иными словами, «сокращённые» формы вещественных литералов вида `5.`, `.77` или `.3E-10` не допускаются.

Вещественные литералы считаются объектами типа `Real` из стандартной библиотеки языка.

Примеры вещественных литералов:

```
123.456
0.99
1.2e-10
77E5
```

Эскейп-последовательности

Эскейп-последовательности (***escape sequences***) – это специальные конструкции, предназначенные для задания символов, которые не имеют специального графического представления. К таким символам принадлежат, в частности, символы конца строки, табуляции и т.д. Вообще говоря, посредством эскейп-последовательности может быть при необходимости представлен любой символ Unicode.

Эскейп-последовательности используются внутри символьных и строковых литералов. Кроме того, их можно использовать для задания отдельных символов в идентификаторах (из числа разрешенных лексическими правилами для идентификаторов).

Эскейп-последовательность начинается с символа обратной наклонной черты, после которой следует либо представление символа, либо его шестнадцатеричный код:

```
Эскейп-последовательность
      : \\
      | \ Кавычка
      | \ Обозначение-символа
      | \ x Шестнадцатеричная-цифра Шестнадцатеричная-цифра
      | \ u Шестнадцатеричная-цифра Шестнадцатеричная-цифра
            Шестнадцатеричная-цифра Шестнадцатеричная-цифра
```

```
Кавычка
     : ' | "

Обозначение-символа
     : t | n | r
```

В данном правиле первая альтернатива обозначает сам символ наклонной черты. Вторая альтернатива используется для обозначения одинарной или двойной кавычки. Эти две формы дают возможность задания соответствующих символов непосредственно внутри символьных и строковых литералов. Три возможных `обозначения-символа` служат для представления форматирующих символов: конструкция `\t` обозначает символ горизонтальной табуляции, `\n` обозначает символ перевода строки, `\r` – символ возврата каретки.

С помощью конструкций, начинающихся с `\x` или `\u`, можно представить любой символ. Вариант с `\x` используется для представления символов ASCII. Две шестнадцатеричные цифры, следующие после буквы, трактуются как числовой код символа. Вариант с `\u` даёт возможность представить числовой код любого символа из набора Unicode.

Примеры эскейп-последовательностей:

```
\n
\xFF
\u0020
```

Символьные и строковые литералы

Символьные литералы предназначены для явного задания одиночных символов. Символьный литерал строится из представления символа, заключённого в одинарные кавычки.

```
Символьный-литерал
     : ' Символ '

Символ
     : Любой-символ-представимый-в-явном-виде
     | Эскейп-последовательность
```

Символьные литералы считаются объектами типа `Char` из стандартной библиотеки языка.

Примеры символьных литералов:

```
'f'             символ буквы f
' '             символ пробела
'\''            символ одиночной кавычки
'\x20'          эквивалентное представление символа пробела
```

Строковые литералы представляют последовательности символов. Строковый литерал строится из последовательности представлений символов, образующих строку. Эта последовательность заключается в двойные кавычки. Каждый символ строки представляется либо своим явным написанием, либо в виде эскейп-последовательности.

```
Строковый-литерал
     : " { Символ } "
```

Строковые литералы считаются объектами типа `String` из стандартной библиотеки языка.

Примеры строковых литералов:

```
""                      // пустая строка
"It's me, o Lord!"
```

```
"\"Title\""             // строка "Title"
"\x20\x20\x20"          // строка, состоящая
                        // из трех пробельных символов
```

Булевские литералы

Булевские литералы задаются посредством идентификаторов `false` и `true`. Эти идентификаторы именуют константные объекты, определённые в контейнере `Boolean` из стандартной библиотеки языка и обозначают два общепринятых значения истинности.

5. Структура программы и основные компоненты

Под *программой* на языке SLang неформально понимается произвольная совокупность *программных единиц*. В качестве программных единиц могут выступать ***контейнеры*** (***units***) и ***подпрограммы*** (***routines***).

```
Единица-компиляции
    : { Директива-использования } { Программная-единица }

Программная-единица
    : Анонимная-подпрограмма | Контейнер | Подпрограмма
```

Контейнер является единой конструкцией как для группирования логически связанных ресурсов (атрибутов, типов и подпрограмм), так и для задания типов данных. Подпрограммы (процедуры и функции) являются вычислительными единицами и служат для вычисления новых значений, для модификации внешнего состояния программы или для преобразования значений из одного типа в другой. Кроме того, подпрограммы могут служить реализациями фиксированного множества *операций* над объектами различных типов; такие подпрограммы называются функциями-операциями.

Строго говоря, понятие *программы* в языке явно не определяется. Программа как самостоятельная сущность возникает в процессе компиляции и объединения (линковки) некоторой совокупности программных единиц.

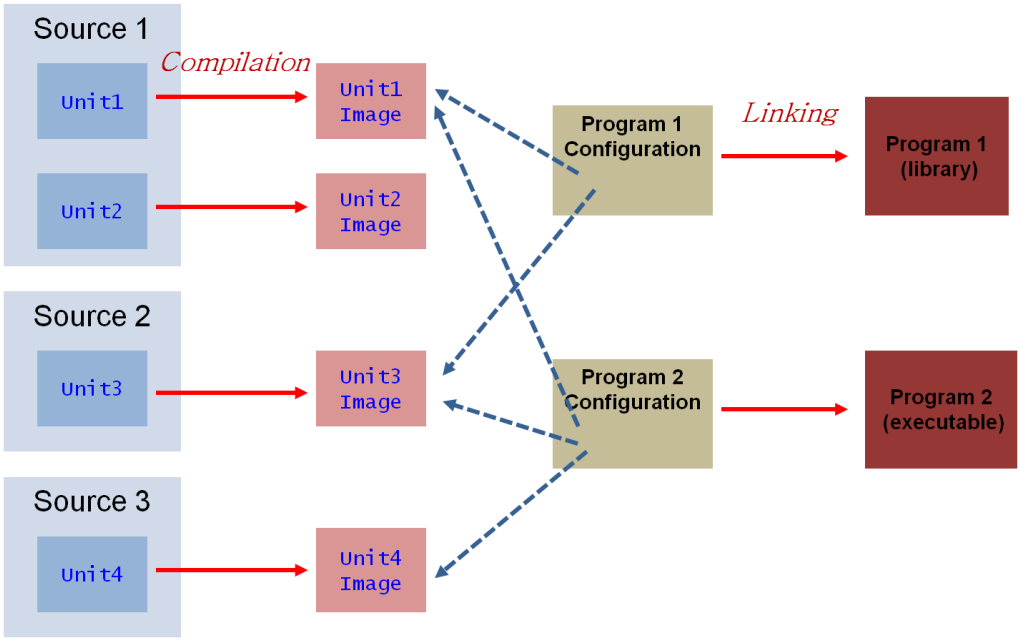
> Множество программных единиц, образующих программу, определяется внешним по отношению к языку способом, например, путём задания так называемого «конфигурационного файла», в котором задаются способы нахождения всех программных единиц, образующих данную программу.
>
> Конкретный способ описания программы в конфигурационном файле может предусматривать либо явное перечисление программных единиц, образующих программу, либо задавать некоторый способ поиска этих единиц.

Как уже говорилось, понятие «исходного файла» или «файла с исходным текстом программы» (source file) в языке не определяется. Конкретные источники исходных текстов программных единиц задаются внешним по отношению к программе способом.

Расположение программных единиц внутри отдельного источника может быть произвольным. Отдельный источник может содержать как единственную программную единицу, так и включать несколько программных единиц. Язык никак не ограничивает структуру исходного текста: в частности, в нём допускается наличие нескольких программных единиц. Конкретное расположение исходного текста программной единицы никак не влияет на её семантику и на семантику единиц, находящихся в том же источнике.

Одна и та же программная единица может входить в качестве составной части в различные программы.

Принципиальная схема конструирования программ из программных единиц представлена на следующем рисунке:



В языке нет разделения программных единиц на части (как, например, partial-классы языка C#); каждая программная единица представляется как единое целое. Исключением служат «внешние» подпрограммы, реализация которых выполнена вне схемы, приведённой выше (и, возможно, на некотором другом ЯП), см. разд. X.

Частным случаем программной единицы может служить простейший вариант, когда она состоит из последовательности объявлений и операторов. Такая конструкция называется анонимной-подпрограммой и определяется следующим синтаксическим правилом:

```
Анонимная-подпрограмма
            : { Объявление-или-оператор }
```

Такая возможность позволяет в рамках языка задавать относительно короткие последовательности действий («скрипты») и тем самым расширить область применения языка. Примером анонимной подпрограммы может служить классическая конструкция «Hello world», которая может быть записана следующим образом:

```
StandardIO.put("Hello world!\n")
```

В предположении, что `StandardIO` представляет собой библиотечный модуль, содержащий, в частности, подпрограммы вывода на печать, такой оператор считается завершённой программой, не требующей для успешной компиляции какой-либо дополнительной информации.

Приведённый выше синтаксис `единицы-компиляции` допускает свободное чередование допустимых элементов. Иными словами, в пределах единицы компиляции контейнеры, подпрограммы, а также отдельные объявления и операторы (или их группы) могут следовать друг за другом в произвольном порядке.

Такое правило облегчает быстрое создание небольших единиц («скриптов»), выполняющих относительно простые функции, но в то же время может способствовать неаккуратному программированию, поощряя создание слабо структурированных и нечитабельных программ. Подобная свобода написания программ с произвольной структурой, предоставляемая языком, может быть ограничена теми или иными руководствами или рекомендациями, которые не являются частью определения языка.

Отсутствие жёстко определённого понятия программы влечёт ещё одну особенность: отсутствие «точки входа» или «стартовой функции», которая в некоторых языках жёстко определяет место, с которого начинается выполнение программы. В языке SLang в качестве стартовой может выступать любая одиночная (невложенная) подпрограмма или инициализатор контейнера. Задание точки входа производится в конфигурационном файле, вместе с перечнем единиц, образующих программу. Параметры окружения доступны в программе посредством соответствующих запросов; множество таких запросов собрано в библиотечный модуль `Environment` (см. главу X).

> Доступ к программному окружению, помимо обращения к подпрограммам из модуля `Environment`, можно было бы реализовать посредством прямой передачи пакета «переменных окружения» в качестве параметра «стартовой» подпрограммы (например, в виде массива строк). Однако такой способ неоправданно сужает доступ к информации об окружении до тела единственной «стартовой» подпрограммы – в то время как запросы к окружению могут производиться из любой точки программы.

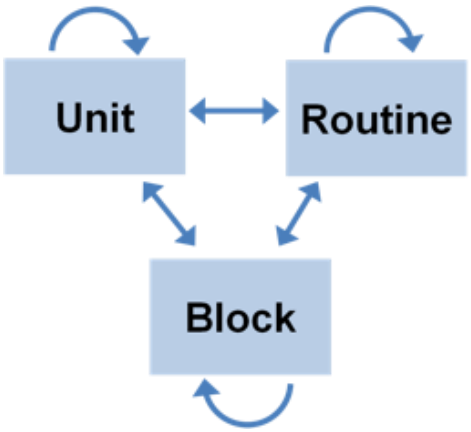
Вложенность программных единиц

Чем больше и сложнее программа, тем более сложной структурой она обладает. В частности, программные единицы могут представлять собой достаточно нетривиальные структурные конструкции, содержащие вложенные конструкции.

Общее правило, принятое в языке SLang, звучит очень просто: 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 подпрограмм. 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.

> Исторически первые относительно развитые языки, начиная с Алгола-60, допускали вложенность подпрограмм. Возможность сохранялась и дальше, вплоть до Паскаля и Ады. Традиция запрета вложенных (nested) подпрограмм идёт от языка C, в котором все функции должны были быть выстроены «в линеечку». Такое упрощающее решение было обусловлено прежде всего соображениями эффективности программ: вызов вложенной функции требовал нескольких дополнительных команд. Однако с сегодняшней точки зрения недопущение некоторого языкового свойства по каким-то «техническим» причинам находится в противоречии с принципом «естественности» языка; в «идеальном» языке не должно быть ограничений на любое сочетание базовых механизмов. Скажем, если в локальном контексте допускаются объявления, то почему ограничивается номенклатура этих объявлений?? Почему в блоке можно объявить, скажем, переменные, константы и типы, но нельзя объявить функцию или класс?..
>
> Авторы языка отдают себе отчёт в том, что произвольная вложенность компонентов программы на практике может привести к её усложнению и снизить читабельность. Тем не менее, представляется, что конкретные ограничения на вложенность, если они будут признаны обоснованными, разумнее оставить руководствам по программированию, нежели фиксировать в языке.
>
> Есть и ещё одно соображение: тенденция включения в «мейнстримные» языки элементов функционального программирования подразумевает более развитые средства манипулирования функциями как значениями, что, в свою очередь, требует более продвинутой трактовки вложенности. Примером может служить язык Scala, в котором поддерживаются и активно используются вложенные и локальные функции.

Таким образом, в языке SLang отсутствуют ограничения на вложенность компонентов. Допустимые отношения вложенности могут быть наглядно представлены следующей схемой, в которой стрелки показывают отношения вложенности конструкций:



Замечание. Стрелка, направленная от контейнера к блоку (которая должна обозначать допустимость вложения произвольной области действия непосредственно в контекст контейнера) в настоящей версии языка не поддерживается.

Заметим также, что вложенность конструкций обычно предполагает определённые правила видимости объектов, содержащихся в таких конструкциях, и правила доступа к ним. Правила видимости и доступа к объектам внутри каждой единицы компиляции обсуждаются в главах, посвящённых этим единицам.

Зависимости между программными единицами

Программные единицы, образующие программу, находятся в определённых взаимосвязях друг с другом. Например, подпрограмма из некоторого контейнера может содержать вызовы подпрограмм или обращения к объектам из другого модуля; любая подпрограмма может создавать объекты и вызывать методы этого объекта, и так далее. Как правило, подобные взаимосвязи задаются посредством составных имён, которые образуются из имени программной единицы и имени сущности из этой единицы, разделённых символами «точка». Например, для обращения к сущности `m` из контейнера `M` используется форма `M2.m`.

Можно сказать, что имена всех программных единиц самого внешнего уровня вложенности, образующих программу, находятся в некотором ***глобальном контексте*** (global context).

Пример. Пусть имеется программная единица – контейнер `Math`, в котором имеется функция с именем `sin`. Тогда обращение к этой функции в простейшем случае может производиться следующим образом:

```
ClientProc is
  ... Math.sin ...
end
```

Имя `Math` будет отыскиваться в глобальном контексте, а имя `sin` – в контексте модуля `Math`. Этот последний контекст в данном случае может быть назван ***локальным контекстом*** (local context) модуля.

Глобальный контекст программы образуется из имён всех программных единиц, которые потенциально доступны для данной программы. Технически, это имена сущностей, объявленный в этой самой программной единице, а также имена программных единиц, располагающихся на носителях, которые известны

компилятору в момент компиляции (например, единицы, содержащиеся в каталогах, имена которых перечислены в конфигурационном файле или просто переданы компилятору в качестве параметров).

> Заметим, что имена программных единиц, объявленных вместе с данной единицей в одном и том же источнике, не имеют каких-либо преимуществ перед именами единиц, заданных в конфигурационном файле. Глобальный контекст образуется как совокупность равноправных единиц.

Тем самым, реализация отношения использования не требует какой-либо специальной языковой конструкции и обеспечивается на уровне системы программирования. Дополнительные сведения об этих аспектах содержится в разделах «Явное задание зависимостей. Директива `use`» и «Контейнеры как модули. Ещё о конструкции `use`». далее.

Пространства имён и составные имена

В некоторых языках имеются средства явного структурирования пространств имён – namespaces. Данный язык не содержит подобных средств; представляется, что введение ещё одного уровня структурирования, в дополнение к программным единицам, неоправданно усложняет структуру программных конфигураций и не является по-настоящему необходимым.

Чтобы уменьшить вероятность конфликтов имён, когда две или более программные единицы, используемые в программе, имеют одинаковые имена, в языке имеется возможность задавать для программных единиц ***составные имена*** (compound names). Пусть, например, имеется модуль с именем `Math`, который содержит специфические для организации-разработчика реализации основных математических функций (`max`, `sin` и т.д.), и имеется высокая вероятность использования такого модуля вместе с другими реализациями аналогичных математических функций. Тогда вместо простого имени `Math` имело бы смысл поименовать этот модуль таким образом, чтобы обеспечить уникальность его имени, например:

```
unit CompanyName.DepartementName.LabName.Math
  ...
end
```

Как видно, составные имена функционально эквиваленты пространствам имён, но имеют то преимущество, что для их использования нет необходимости явно вводить специальное понятие. Разработчик единицы не связан специальным синтаксисом и не должен загромождать программу, окружая свою единицу скобками с заголовком. С другой стороны, в целях большей систематичности именования программных единиц разработчик (отдельный программист или организация) может следовать определённым правилам образования составных имён.

Замечание. Составные имена допускаются только для программных единиц – то есть, для контейнеров и подпрограмм самого внешнего уровня. Вложенные и локальные компоненты должны иметь имена, состоящие из одиночных идентификаторов.

> Разумеется, составные имена (точно так же, как и механизм пространств имен) не разрешают в полной мере проблемы с конфликтами имен. Например, если два разных программиста написали контейнер `Math` и не сделали эти имена составными, или сформировали для него одинаковое составное имя, то язык не предоставляет механизма для разрешения такого конфликта имен. Подобные конфликты могут разрешаться на уровне управления программной конфигурацией.

Явное задание зависимостей. Фраза `use`

Как правило, поиск используемой программной единицы по глобальному контексту достаточен для корректного разрешения имён. Но в некоторых случаях бывает необходимо явное задание зависимостей. Имеются две причины задавать зависимости явно.

Во-первых, в ряде случаев удобно использовать сокращённое имя программной единицы внутри использующего контекста. Для этих целей можно использовать специальную фразу (директиву) со служебным словом **`use`**, которая считается частью заголовка. Примером может служить следующий фрагмент программы:

```
// Для длинного составного имени предлагается
// короткий синоним
unit Client
    use CompanyName.DepartmentName.LabName.Math as Math
    ... Math.sin ... // полное имя функции sin
end
```

Другая ситуация возникает в случае, когда необходимо снять потенциальный конфликт имён для нескольких одноимённых единиц, используемых в данной единице, например:

```
unit Client
    use CompanyName.DepartmentName.LabName.Math as Math1
    use ForeignCompany.Math as Math2
    ... Math1.sin ...   // Функция sin из контейнера Math1
    ... Math2.sin ...   // Функция sin из контейнера Math2
end
```

Детальное описание семантики директивы использования будет обсуждаться в разделе «Контейнеры как модули. Ещё о конструкции `use`».

Синтаксис директивы использования описывается следующими правилами:

```
Директива-использования
    : use Используемая-единица { , Используемая-единица }

Используемая-единица
    : Имя-программной-единицы [ as Короткое-имя-единицы ]

Имя-программной-единицы
    : Составное-имя

Составное-имя
    : Идентификатор { . Идентификатор }

Короткое-имя-единицы
    : Идентификатор
```

Замечания.

1. Предыдущие рассмотрения и примеры показывают, что в использующем контексте достаточно использовать простое имя. Так, если для единицы задана фраза **`use`** `Math`, то для именования доступной сущности из этой единицы достаточно использовать простое имя, например, `sin`. Тем не менее, в языке принято правило, согласно которому все сущности, импортируемые из некоторой единицы, должны именоваться посредством её полного (составного) имени, то есть, `Math.sin`, даже если компилятор может однозначно идентифицировать простое имя.

2. Как правило, циклические зависимости программных единиц друг от друга не допускаются. В некоторых контекстах они считаются ошибкой проектирования и

детектируются компилятором; в других случаях зависимости считаются допустимыми.

Конкретные случаи зависимостей и их интерпретация рассматриваются в разделах, относящихся к природе таких зависимостей.

6. Контейнеры

Контейнеры являются одним из базовых понятий языка и в то же время представляют основной строительный блок для формирования программ на языке SLang. Как уже говорилось во введении, ***контейнер*** можно определить как поименованную совокупность атрибутов и подпрограмм, которая может быть параметризована либо типом, либо константой, 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.

Контейнер можно рассматривать как определение множества данных и операций над ними – то есть, как задание некоторого **типа**. Тем самым, можно определить объект, тип которого будет 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*. Контейнер представляет собой весьма общее понятие, а та или иная конкретная «роль» контейнера определяется *способом его использования*.

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

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

Наиболее естественный способ использования такого контейнера – это его применение в качестве **модуля**: он должен явно включаться в клиентскую программу, в процессе ее работы он существует в единственном экземпляре, независимо от числа его подключений. 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 понятия контейнера в языке SLang можно сформулировать следующим образом: при сохранении единой формы задания контейнеров произвольной природы (модуль, класс, тип) возможны различные способы их использования. Иными словами, один и тот же контейнер может выступать в различных «ипостасях» в зависимости *от контекста его использования*.

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 процедуру

инициализации («конструктор») без параметров, или не иметь никаких процедур инициализации.

Синтаксис объявления контейнера выглядит следующим образом:

```
Объявление-контейнера
        :   [ Спецификатор-контейнера ]
          unit Имя-контейнера [ FormalGenerics_]
            { Директива-контейнера }
          is
            Тело-контейнера
            [ invariant Список-предикатов ]
          end [ Имя-контейнера ]

Спецификатор-контейнера
        : ref | val | concurrent | abstract

Директива-контейнера
        : Директива-наследования
        | Директива-использования

Директива-наследования
        : extend Базовый-контейнер { , Базовый-контейнер }

Базовый-контейнер
        : [ ~ ] UnitTypeName

Тело-контейнера
        : { Объявление }
```

В этой главе мы рассмотрим использование контейнеров для организации модульной структуры программ. «Классовая» природа контейнеров и соответствующее ей понятие типа будут обсуждаться в последующих главах.

Контейнеры как модули

Модульный принцип – один из наиболее распространённых и практически удобных способов организации программы. Согласно этому подходу, программа представляется как совокупность синтаксически обособленных фрагментов, заключающих ту или иную завершенную функциональность. Модули взаимодействуют друг с другом посредством явно определенных интерфейсов, фактически представляющих собой правила обращения к тем или иным ресурсам модуля со стороны других модулей.

Модульный принцип позволяет снизить общую сложность программной системы, представив её в виде композиции относительно простых коллекций логически связанных ресурсов (данных и/или алгоритмов).

В процессе выполнения программы каждый модуль представлен как неизменная сущность, созданная в единственном экземпляре. Создание модулей производится либо на стадии формировании программы (статически), либо динамически – системой времени выполнения (в начале работы программы или в некоторый момент её выполнения). Во всех случаях программист не имеет возможности явно управлять этим процессом.

В качестве примера рассмотрим контейнер `Math`, упомянутый в предыдущем разделе. Этот контейнер представляет собой коллекцию функций, реализующих типичные математические операции. Другие программные единицы могут использовать ресурсы этого контейнера для собственных целей. При этом, очевидно, все такие программные единицы будут использовать («разделять») один и тот же контейнер, который существует в процессе выполнения программы

в единственном экземпляре. Поэтому естественно считать такой контейнер *модулем*.

Если в некоторой программной единице (подпрограмме или в другом контейнере) необходимо использовать ресурсы контейнера `Math`, то можно сделать это непосредственно, не прибегая к каким-либо специальным средствам, например:

```
unit Client
    ...
    foo() is
        ...
        x is Math.sin(x+Math.pi)
        ...
    end foo
end
```

При этом следует отметить два обстоятельства:

1. Доступ к ресурсам контейнера `Math` производится посредством обычной точечной нотации, в которой задаётся имя контейнера (в общем случае составное имя) и простое имя компонента контейнера.

2. Контейнер `Math` должен находиться в глобальном контексте использующей программной единицы. Это означает, что точное местонахождение исходного текста контейнера или его кода `Math` или его кода должно быть задано в конфигурационном файле или каким-либо ещё «внешним» по отношению к программе способом.

Более точно, семантика использования контейнера как модуля заключается в следующем. В неопределённый момент времени, предшествующий первому обращению к ресурсам контейнера создаётся экземпляр контейнера `Math`. После этого первое и все последующие обращения к ресурсам `Math` будут ссылаться на экземпляр `Math`.

Это правило распространяется на все экземпляры (объекты) всех контейнеров программы, а также на все standalone-подпрограммы. Иными словами, все эти объекты будут *разделять один и тот же экземпляр* `Math`.

Ещё о конструкции `use`

Как уже говорилось ранее, вообще говоря, нет необходимости явно задавать использование некоторого контейнера в качестве модуля. Компилятор в состоянии идентифицировать такое использование по вхождению имени модуля в составе обращения к какому-либо его свойству (ресурсу). Тем не менее, как описывалось в разделе «Явное задание зависимостей. Директива `use`», в языке имеется конструкция явного задания зависимостей, которая не является обязательной, но предназначена для упрощения использования составных имён контейнеров и для снятия возможных неоднозначностей. Приведём ещё раз один из примеров из упомянутого раздела:

```
// Для длинного составного имени предлагается
// короткий синоним
unit Client
    use CompanyName.DepartmentName.LabName.Math as Math
    ... Math.sin ... // полное имя функции sin
end
```

В этом примере контейнер `Math` снабжён составным именем, которое позволяет с достаточно высокой степенью вероятности обеспечить его уникальность. В то же время обращение к ресурсам этого модуля по его полному имени может сделать

программу слишком громоздкой и нечитабельной. Использование фразы **use** позволяет сократить составное имя, предоставив его более короткий синоним.

Заметим, что синоним составного имени, введённый фразой **use**, действует только внутри области действия того контейнера, в заголовке которого задана эта фраза. Следующий схематический пример показывает ограниченность действия фразы **use**:

```
// Фраза use действует только внутри контейнера Client1…
unit Client1 use CompanyX.Math as Math
    ...
    x is Math.sin(x+Math.pi)
    ...
end

// …И не действует внутри другого контейнера
unit Client2
    ...
    x is Math.sin(x+Math.pi)
      // ошибка компиляции: нет контейнера с именем Math
    y is CompanyX.Math.sin(x+CompanyX.Math.pi)
      // правильно: контейнер CompanyX.Math присутствует
      // в глобальном контексте.
    ...
end
```

В некоторых, достаточно типичных, случаях во всех программных единицах, заданных в одном источнике (исходном тексте), должны использоваться ресурсы одного и того же контейнера-модуля. Так, свойства используемого в примерах выше контейнера `Math` могут использоваться в нескольких отдельно-стоящих подпрограммах, размещённых в одном источнике.

В подобных случаях было бы слишком громоздко повторять одну и ту же фразу **use** в заголовке каждой такой подпрограммы. Поэтому в языке имеется возможности задания **глобальной** фразы **use**, не привязанной к конкретной программной единице, а относящейся ко всем единицам из данного источника.

```
// Действие глобальной фразы use распространяется
// на все программные единицы из данного источника
use CompanyName.DepartmentName.Math as Math

quadRoots(a, b, c: Real): (Real, Real) is
    ...
    // Использование модуля Math
    ...
end

complexMult(a, b: Real): Real is
 ...
    // Использование модуля Math
...
end
```

Фраза `use`: полная семантика

Использование фразу **use**, показанное в примерах из предыдущего раздела, служит, скорее, целям большего удобства создания программ путём сокращения повторяющихся фрагментов. На самом деле, конструкция **use** обладает существенно б**о**льшими возможностями.

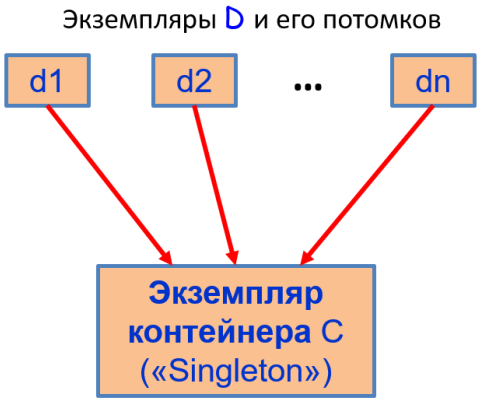
Рассмотрим простой пример. Пусть имеется контейнер `D`, в заголовке которого задана фраза **use**, указывающая на некоторый другой контейнер `C`:

```
unit D use C
    ...
end
```

Семантика «использования» контейнера C заключается в том, что все объекты типа D, возникающие в процессе выполнения программы, а также объекты типов, производных от D, имеют доступ *к одному и тому же объекту* типа C. Этот последний объект создается до первого обращения к его атрибутам или свойствам из объектов D или из объектов его производных типов.

Схематически это может быть проиллюстрировано следующей схемой:

```
unit D use C
    ...
end D
```

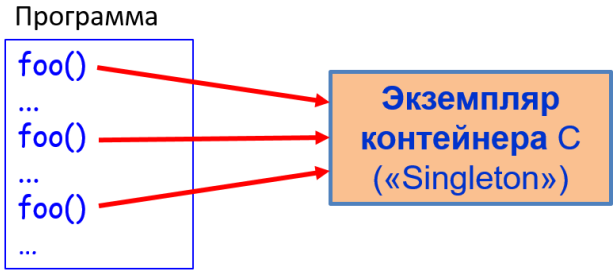


Аналогичное правило применяется в случае задания фразы **use** в заголовке подпрограммы:

```
foo() use C is
    ...
end
```

В процессе выполнения подпрограммы foo (при каждом ее вызове) она имеет доступ *к одному и тому же* объекту типа C; этот объект создается системой перед самым первым обращением к его свойствам и атрибутам из подпрограммы foo. Наглядно это можно представить таким образом:

```
foo() use C is
    ...
end
```



Заметим, что в двух последних примерах для объектов типа D и его потомков, с одной стороны, и для вызовов подпрограммы foo, с другой стороны, будут созданы *отдельные* экземпляры контейнера C.

Принятые в языке правила, связанные с использованием **use**, предоставляют достаточно развитые средства структурирования программ и данных, в частности, избавляют от необходимости так называемых «статических» членов контейнеров.

7. Контейнеры как типы

В предыдущих главах в качестве примера мы рассматривали контейнер `Math`, содержащий программные реализации математических функций. Этот контейнер использовался в качестве модуля, ресурсы которого либо включались в состав некоторой использующей программной единицы, либо становились доступными для нескольких единиц из одного источника.

Наряду с таким механизмом использования контейнеров, имеется возможность трактовать контейнер *как тип*. Продолжим наши рассуждения, взяв в качестве примера тот же контейнер `Math`.

Вполне вероятно, что возникнет необходимость расширить функциональность `Math` – например, за счёт 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 традиционный для многих языков программирования *класс*.

Продолжая эту логику, естественно предположить, что 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`, который инициализируется значением типа `Math`, и свойства которого 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
```

Типы-значения и ссылочные типы

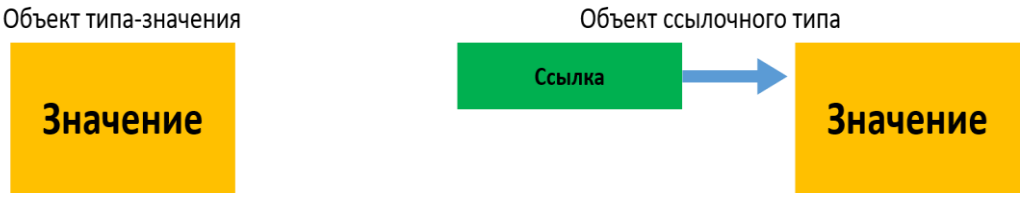
В некоторых современных языках программирования (прежде всего, в Java и C#) все типы подразделяются на две большие категории – типы-значения (value types) и ссылочные типы (reference types).

> Такое деление возникло в попытках избавиться от низкоуровневого и ненадёжного понятия указателя, широко используемого в языках предыдущего поколения, таких как C и C++.

Под ***типом-значением*** понимается тип, объекты которого представляют непосредственно сами себя. Например, объекты таких типов, как `Integer` или `Real`, непосредственно содержат, соответственно, целочисленные или вещественные значения.

В отличие от типов-значений, объекты ***ссылочных типов*** всегда представляются в виде пары, которая состоит из самого значения и некоторого представителя этого значения – *ссылки*.

Схематически это можно представить в виде следующей картинки:



Связь между ссылкой и значением для ссылочных типов не является жёсткой; иными словами, в процессе работы программы ссылка может быть перенацелена на другое значение того же типа или оказаться не нацеленной ни на какое значение. Подробнее этот механизм будет рассматриваться ниже.

В языке SLang поддерживается деление на типы-значения и ссылочные типы, однако эта категоризация реализована в более гибком варианте по сравнению с другими языками. В SLang можно привязывать категорию как к самому типу, так и к *объектам* этого типа.

Сначала рассмотрим свойство «ссылочности» применительно к типам-контейнерам (и, соответственно, оно окажется свойством всех объектов данного типа). Так, следующее объявление

```
value unit ValUnit
    av is T
    ...
end ValUnit
```

вводит контейнер `ValUnit`, все объекты которого по умолчанию являются объектами типа-значения. Это означает, что объявление вида

```
v is ValUnit
```

приводит к созданию объекта-значения типа `ValUnit`. Переменная `v` непосредственно представляет созданный объект, и атрибуты объекта доступны посредством точечной записи, в которой перед точкой указывается имя переменной `v`, например, `v.av`.

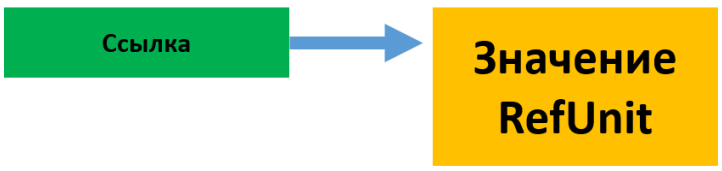
В то же время объявление

```
ref unit RefUnit
    ar is T
    ...
end RefUnit
```

определяет контейнер `RefUnit`, все объекты которого по умолчанию являются объектами ссылочного типа. Это означает, что объявление вида

```
r is RefUnit
```

приводит к созданию *пары*: ссылки на объект типа `RefUnit` и самого значения этого типа:

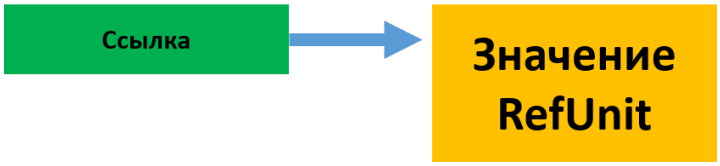


Тем самым, имя `r` обозначает не сам объект типа `RefUnit`, а ссылку на него. Однако, доступ к значению этого ссылочного типа производится указанием имени переменной `r`, в точности так же, как и доступ к объектам типов-значений: `r.ar`.

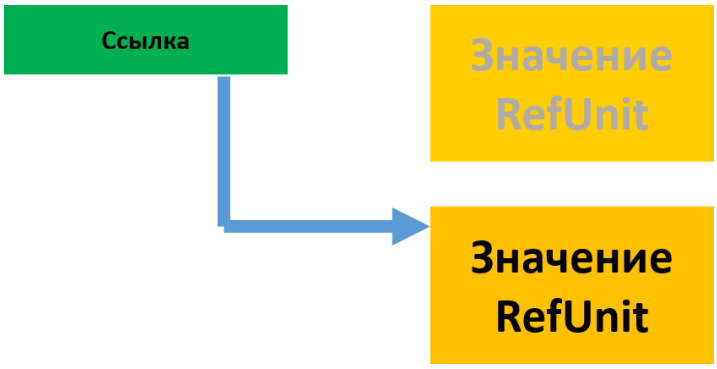
Таким образом, различие между типами-значениями и ссылочными типами лежит больше в сфере реализации; внешне форма доступа к объектам всех типов выглядит одинаково. Тем не менее, имеются некоторые семантические отличия. Рассмотрим такую последовательность двух операторов:

```
r is RefUnit
...
r := RefUnit()
```

Первый оператор вводит переменную ссылочного типа и инициализирует ее объектом этого типа. Конфигурация памяти, соответствующая состоянию после первого оператора, выглядит следующим образом:



Второй оператор присваивает переменной (фактически, ссылке) `r` новое значение. Как следствие такого присваивания, значение, на которое до этого ссылалась переменная `r`, становится недоступным:



Доступ к атрибутам и свойствам контейнера

Пока только основные позиции:

- Все атрибуты контейнеров **доступны по чтению** – как для клиентов данного контейнера, так и для контейнеров-наследников.
- Если необходимо ограничить доступ к атрибуту, следует объявить его со спецификатором `hidden`. В этом случае атрибут будет недоступен для клиентов контейнера, но доступен для его потомков.
- Если необходимо ограничить доступ к атрибуту самим контейнером, в котором он объявлен, то его следует объявить со спецификаторами `hidden final`.
- Все атрибуты контейнера **закрыты по записи** для клиентов контейнера. Непосредственный доступ по записи для атрибутов контейнера возможен только изнутри самого контейнера.

- Если необходимо обеспечить изменение состояния контейнера (то есть, изменить значение какого-либо его атрибута), то следует либо определить подпрограмму контейнера, которая бы изменяла это значение, либо определить собственную версию операции присваивания для типа данного атрибута.

Инициализаторы контейнеров

Инициализатор контейнера – это специальная подпрограмма, которая служит для установки начального состояния объекта контейнера, в котором она задаётся. Инициализатор вызывается при создании объекта.

Синтаксически инициализатор определяется аналогично подпрограммам, но в качестве имени используется специальное служебное слово `init`.

В одном контейнере может быть задано несколько инициализаторов, которые в этом случае должны различаться числом и типом (типами) своих параметров. Можно считать, что такие инициализаторы задают несколько способов создания объекта.

Ниже показан пример контейнера с двумя инициализаторами.

```
unit Point
    x, y: Real
    init() is
        x := 0.0
        y := 0.0
    end
    init(a,b: Real) is
        x := a
        y := b
    end
    moveHor(d: Real) => x := x+d
    moveVer(d: Real) => y := y+d
end
```

Контейнер `Point` моделирует точки на плоскости. Каждая точка (объект типа `Point`) характеризуется своими координатами. Каждый объект типа `Point` при его создании должен быть инициализирован посредством вызова одного из двух инициализаторов. Первый инициализатор устанавливает начальную позицию точки в начале системы координат, второй инициализатор задает конкретные начальные координаты.

Существует несколько способов объявления объекта данного типа. Пример наиболее полного варианта объявления выглядит следующим образом:

```
point: Point is Point.init(1,2)
```

В таком объявлении явно указывается тип нового объекта и выражение (явный вызов инициализатора), задающее начальное состояние этого объекта.

Для упрощения практического программирования некоторые компоненты такого объявления могут быть опущены без потери информации и без изменения его семантики. Так, тип объявляемого объекта может быть автоматически выведен из типа инициализирующего выражения, и потому его можно опустить:

```
point is Point.init(1,2)
```

Выражение-инициализатор также можно сократить, опустив служебное слово `init`:

```
point is Point(1,2)
```

Кроме того, если предполагается задать начальное значение объекта неявно, то такое объявление можно записать следующим образом:

```
point: Point      // начальное значение для point будет
                  // задано инициализатором без параметров
```

или

```
point is Point    // начальное значение для point задается
                  // выражением, то есть, вызовом
                  // инициализатора без параметров;
                  // тип point выводится из типа выражения
```

В обоих случаях для задания начального состояния объекта `point` будет вызван инициализатор без параметров.

Инвариант контейнера

Наследование

Наследование как вид отношений между контейнерами дает возможность расширять возможности новых контейнеров, используя уже имеющиеся и доступные свойства существующих.

Определение конформности

Множественное наследование

> Основные проблемы, которые возникают при реализации наследования, – это конфликт имен, неоднозначность версий функции при полиморфном присваивании, а также согласование статусов экспорта функций. Отмеченные проблемы возникают в основном при множественном наследовании, поэтому во многих современных языках возможности наследования ограничивают единичным наследованием, предлагая в качестве паллиативного решения понятие интерфейса (C#, Java) или «протокола» (Swift [7]).
>
> Можно, однако, сохранить механизм множественного наследования со всеми его преимуществами, если не ставить задачей контроль правильности всего графа наследования для всех свойств всех контейнеров - как делается, например, в языке Eiffel, который имеет мощный, но непривычный аппарат адаптации свойств при наследовании. Вместо этого можно ограничиться решением более простой задачи: проверки на наличие однозначного разрешения *обращения к свойству* контейнера (validity of the call).

Язык SLang поддерживает модель множественного наследования, согласно которой любой контейнер может быть определён как *производный* от некоторого числа других контейнеров, которые называются *базовыми* для первого контейнера. Считается, что производный контейнер включает все свойства базовых контейнеров, а также собственные (определённые в нём самом) свойства. При этом некоторые свойства базовых контейнеров могут быть переопределены в производном контейнере.

<Примеры>

Наследование и инициализаторы

Основное:

- Инициализаторы для базовых контейнеров вызываются явно.

Наследование и инварианты контейнеров

- Как определяется «итоговый» инвариант производного контейнера.

Множественное наследование: принципы разрешения неоднозначностей

Данный раздел содержит обсуждение преимуществ предложенной альтернативы. Рассмотрим схематический пример контейнера, в котором заданы две одноименные подпрограммы с различными сигнатурами.

```
unit Base
    foo(signature1)
    foo(signature2)
end Base
```

Общая задача может быть сформулирована так: является ли правильным контейнер `Base`? Иными словами, при каком соотношении сигнатур он непротиворечив? Вместо этой общей задачи поставим более конкретную задачу, которая исходит из *использования* контейнера в качестве *типа*. Пусть имеется обращение к подпрограмме `foo`:

```
b is Base
b.foo(arguments)
```

Мы хотим проверить, является ли корректным обращение к `b.foo` с некоторыми аргументами. Такое обращение будет правильным, если в `Base` есть одна или несколько подпрограмм с именем `foo`, и типы аргументов вызова имеют однозначное разрешение среди сигнатур всех свойств `foo`. Если типы всех аргументов конформны только сигнатуре *signature1*, то перед нами обращение к первой версии `foo`, если только сигнатуре *signature2* – ко второй `foo`. Если же аргументы вызова конформны обеим сигнатурам, то фиксируется неоднозначность, что трактуется компилятором как ошибка. При этом сам контейнер `Base` с набором указанных подпрограмм *не считается некорректным*. Что же касается самого контейнера `Base`, то единственная проверка, которая представляется для него необходимой, – контроль двух сигнатур на идентичность.

Механизм перегрузки (overloading) может использоваться также для разрешения конфликтов по именам при наследовании. Рассмотрим, как это может работать, на следующем примере.

```
unit Derived extend Base
    foo(signature3)
    override foo(signature4)
end B
```

Контейнер `Derived` вводит *новую* подпрограмму `foo` с сигнатурой *signature3*. Единственное ограничение на *signature3* заключается в требовании неидентичности с *signature4*, определенной в этом же контейнере.

В то же время наличие префикса **`override`** для `foo` с сигнатурой *signature4* приводит к перекрытию (overriding) *обеих* `foo` из базового контейнера `Base`, при условии, что *signature4* конформна обеим сигнатурам *signature1* и *signature2*. Если *signature4* конформна только одной `foo` из базового контейнера, то другая подпрограмма наследуется в `Derived`.

Иными словами, основой механизма разрешения конфликтов перекрытия служит конформность сигнатур. Рассмотрим несколько более сложный пример.

```
unit T1 ... end
unit T2 extends T1 ... end
```

```
unit T3 extends T1 ... end
unit T4 extends T2, T3 ... end

unit Base
    foo(arg1: T1; arg2: T2)
    foo(arg1: T1; arg2: T3)
end Base

unit Derived extends Base
    foo (arg1: T4)
    override foo (arg1: T4; arg2: T4)
end Derived
```

В контейнерах `Base` и `Derived` имеются по две подпрограммы `foo` с различными сигнатурами.

> В последующих примерах нотация вида `T()` обозначает создание нового объекта данного контейнера (то есть, в данном случае он рассматривается как тип). При этом вызывается процедура инициализации контейнера без параметров, если она имеется. В такой нотации можно опускать пустые круглые скобки. Кроме того, перед именем контейнера можно поставить служебное слово **`new`**, чтобы явно подчеркнуть создание объекта (экземпляра), например, **`new`** `T`. Наконец, допускается конструкция явного вызова процедуры инициализации экземпляра: `T.init(`*`arguments`*`)`. Все эти варианты синтаксиса считаются семантически эквивалентными, однако здесь для простоты используется только нотация, указанная первой.

Рассмотрим различные случаи использования подпрограмм `foo`.

```
var b is Base
```

Создается объект `b`, типом которого считается контейнер `Base`. При создании этого объекта производится вызов процедуры инициализации из `Base`.

```
b.foo(T1(),T2())
```

Для данного вызова разрешение для подпрограммы `foo` однозначно: перед нами обращение к первой `foo` из `Base`.

```
b.foo(T1(),T3())
```

Этот вызов однозначно разрешается как обращение ко второй `foo` из `Base`.

```
b.foo(T1(),T4())
```

В этом вызове возникает неоднозначность: могут быть вызваны обе версии подпрограмм `Base.foo`.

Пусть мы присвоили `b` новый объект производного типа `Derived` и вызвали для этого объекта подпрограмму `foo`:

```
b := Derived()
b.foo(T1(),T2())
```

Этот вызов содержит однозначное обращение к первой `foo`. При выполнении будет вызвана версия `foo` из `Derived` с сигнатурой `(T4,T4)`.

> Приведенный случай может служить иллюстрацией так называемых «cat calls»: нарушением типизации вследствие ковариантного переопределения сигнатур и полиморфного присваивания. При проведении полной проверки корректности данного вызова (на уровне всей собираемой системы) он будет отвергнут компилятором.

Продолжим наше рассмотрение. Вызов вида

```
b.foo(T1(),T3())
```

подразумевает однозначное обращение ко второй `foo` из `Base`. При выполнении, как и в примере выше, будет вызвана версия `foo` из контейнера `Derived` с сигнатурой `(T4, T4)`, перекрывающая «базовую» версию.

С другой стороны, вызов вида

```
b.foo(T1(),T4())
```

приводит к неоднозначности: обе версии `foo` конформны аргументам.

В заключение данного раздела отметим, что реализованный в SLang механизм разрешения неоднозначностей в случае множественного наследования, основанный на анализе полиморфного использования, позволяет не решать общую теоретическую задачу проверки правильности графа наследования, а просто решать конкретные ограниченные задачи, с которыми сталкивается программист-практик, оставаясь только с одной концепцией наследования для построения повторно используемого ПО.

8. Обобщённые типы

```
FormalGenerics
        : [ Обобщенный-параметр { , Обобщенный-параметр_} ]

Обобщенный-параметр
        : Обобщенный-типовой-параметр
                  [ Уточнение-типового-параметра ]
        | Обобщенный-нетиповой-параметр : Тип

Уточнение-типового-параметра
        : -> Тип [ init [ ( [ Тип { , Тип } ] ) ] ] ]

Обобщенный-типовой-параметр
        : Идентификатор

Обобщенный-нетиповой-параметр
        : Идентификатор
```

- Допускается существование контейнеров с одним и тем же именем, если один из них – обобщенный, а другой – обычный.
- Допускаются объявления нескольких обобщенных контейнеров с одним и тем же именем. В этом случае они должны различаться по числу и/или видам формальных параметров.

9. Параллельное выполнение

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 параллельные 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 a 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
```

10. Другие механизмы образования типов

Константные объекты

Понятие константных объектов позволяет поднять уровень абстракции на чуть более высокий уровень, чем просто пользование константами и следующий шаг – это понимание, что набор констант – это набор константных объектов некоторого типа. И в общем случае тип может быть любой. Таким образом, любой юнит на базе, которого порождается тип может задать все константные объекты данного юнит типа. Для целого типа это задается следующим примером

```
val unit Integer
    const
       Platform.minInteger .. Platform.maxInteger
    end
end Integer
```

Таким образом, запись `Integer.1` есть полное наименование константного объекта `1`, для того чтобы не писать префикс имени юнита необходимо включить константы в тот юнит или фрагмент кода, где это необходимо

```
abstract unit Any
use const Integer, Real, Boolean, Character
    ...
end
```

Вот такая конструкция **`use const`** позволяет использовать константные объекты из указанных юнитов без префиксов имен юнитов в данном юните и во всех его потомках. Таким образом, константы, к которым мы так привыкли можно использовать без префиксов в любом юните.

```
unit WeekDay
    const
        Monday, Tuesday, Wednesday, Thursday, Friday,
        Saturday, Sunday
    end
end
```

Теперь можно работать с константами типа `WeekDay`

```
day: WeekDay is WeekDay.Monday
if day is
    WeekDay.Monday .. WeekDay.Friday:
        StandardIO.putString (“Work day – go to the office!\n”)
    WeekDay.Saturday, WeekDay.Sunday:
        StandardIO.putString (“WeekEnd – do what you like!\n”)
end
```

А для того, чтобы избежать обязательного префикса, достаточно написать **`use const`** `WeekDay` и после этого тот же самый пример станет короче, но не менее выразителен

```
use const WeekDay
day: WeekDay is Monday
if day is
    Monday .. Friday: StandardIO.putString (“Work day – go to
the office!\n”)
```

```
    Saturday, Sunday: StandardIO.putString ("WeekEnd – do what you like!\n")
end
```

А так как юнит может иметь процедуру инициализации, то пример более общего случая будет выглядеть так

```
unit A
        const A1.init, A2.init, A3.init end
        init is end
        init (arg: T) is end
        init (arg1: T1; arg2: T2) is end
end
```

И теперь уже можно использовать тип A и с константные объекты данного типа A1, A2 и A3.

Например, таким образом

```
use const A
foo (A1)
foo (argument: A) is
        if argument
                A1: // Некоторое действие(я)
                A2 .. A3: // Другое действие(я)
        end
end
```

Понятие константных объектов является более общим, чем перечислимые типы и позволяет поддержать привычный функционал перечислимых типов и распространить возможность сравнение по образцу (pattern matching) для любых типов.

Несколько слов о наследовании и константных объектах. Так как каждый константный объект имеет тип, который соответствует типу текущего юнита, то при наследовании есть две схемы – первая, что автоматически происходит переопределение типа константных объектов по схеме as this, а вторая что ничего не происходит и в потомке они имеют тот же тип, что и в родителе (базовом юните). Первая схема сразу поднимает вопрос о валидности обращений к процедурам инициализации и чтобы не решать данный вопрос предлагается использовать вторую схему – тип константного объекта задается один раз при описании и при наследовании не меняется. Тогда получается, что при наследовании список константных объектов производного юнита (потомка) не является расширением списка объектов базового (родителя). По сути это два независимых списка константных объектов. Что не дает возможности присвоить Integer.1 наследнику юнита Integer. Что просто отвратительно!!! Давайте рассмотрим эти сложности на пример, чтобы прочувствовать, что отношение наследования юнитов прямо противоположно понятию расширения списка константных объектов для юнита.

```
unit B extend A
const B1, B2, B3 end
end
```

```
use const B
a1 is A1 /* Константный объект наследован юнитом B из юнита A и, соответственно
доступен после включения констант из юнита B в текущий контекст и тип у A1 – это A*/
b1 is B1 // B1 доступен и тип его B
a1 := b1 // тип b1 конформен типу a1 – присваивание возможно
```

Основной вопрос встает является ли список константных объектов юнита B расширением списка для A? Другими словами можем ли мы написать так

```
while x in A1 .. B3 loop
if x is
A: // Если тип x A, то
B: // Если тип x B, то
else // Иначе
end
end
```

Так как мы подразумеваем расширение списка константых объектов при наследовании справа, то тип x должен быть типом A автоматически выведенным компилятором и мы модем спокойно статически обращаться к свойствам (подпрограммам и атрибутам), которые доступны из описания A, а вот для того чтобы понять, что x будет иметь динамический тип B, нам нужна динамическая проверка типа, что и делает условный оператор в теле цикла. Т.е. проверка на динамически тип A лишняя, так же как else часть. Таким образом, пример, который так хотелось написать для дней недели при помощи двух юнитов, может выглядеть следующим образом

```
unit WorkDay
const Monday, Tuesday, Wednesday, Thursday, Friday end
end

unit WeekendDay extend WorkDay
const Saturday, Sunday end
end

use const WorkDay
isWorkDay (day: WorkDay): Boolean is
if day is
Monday .. Friday: return True
else return False
end
end
```

Но при таком наследовании получается, что юнит выходной день содержит в себе константы рабочих дней с понедельника по пятницу, что не совсем логично. Отсюда следует, что другая схема наследования более логична

```
abstract unit WeekDay
isWorkDay: Boolean is abstract end
isWeekend: Boolean is abstract end
end

unit WorkDay extend WeekDay
const Monday, Tuesday, Wednesday, Thursday, Friday end
override const isWorkDay: Boolean is True
```

```
override const isWeekend: Boolean is False
end

unit WeekendDay extend WeekDay
const Saturday, Sunday end
override const isWorkDay: Boolean is False
override const isWeekend: Boolean is True
end
```

При такой схеме все хорошо, кроме того что константы дней недели лежат в двух разных списках. И тогда приходится создавать несколько искусственный юнит, куда попадают оба списка констант

```
unit WeekDays use const WorkDay, WeekendDay
end
```

И вот теперь у нас есть все константы дней недели, но они находятся не в юните WeekDay, а в юните WeekDays при чем типы констант для рабочих дней WorkDay, а для выходных WeekendDay, что логично, но непривычно.

```
use const WeekDays
while x in Monday .. Sunday loop // тип x будет WeekDay!
if x is
Monday .. Friday: StandardIO.put (“Рабочий день”)
Saturday, Sunday: StandardIO.put (“Выходной”)
else StandardIO.put (“А сюда мы никогда не попадем □”)
end
end
```

Таким образом – списки констант при наследовании можно расширять, можно объединять, но надо помнить про типы этих объектов и правильно этим пользоваться.

```
unit WorkDay
   const Mon, Tue, Wed, Thu, Fri end
end

unit WeekendDay
   const Sat, Sun end
end

unit WeekDay extend WorkDay, WeekendDay
end

d: WeekDay is WorkDay.Mon
```

Функциональные типы

```
a: routine(T1,T2):T3
b: routine(T1,T2)
c: (T1, T2):T3
foo(T1, T2):T3 is ... end
```

```
a := foo              // ok
a := foo(T1(),T2())   // type mismatch
c := foo              // ok
c(T1(),T2())          // ok; the same as foo(T1(),T2())
b := foo              // type mismatch
```

Опциональные типы

Кортежи

Понятие кортежа и соответствующий языковой механизм появились в промышленных языках программирования сравнительно недавно, однако быстро стали распространенным инструментом программирования.

Язык Scala содержит простой механизм кортежей, который служит только для агрегации нескольких разнородных элементов в единую конструкцию. Такие новые языки, как Rust и Swift включают более развитую поддержку этого понятия.

В языке SLang кортежи введены в сравнительно ограниченном варианте, чтобы не отпугнуть программистов излишней сложностью относительно нового механизма. Будучи, как мы увидим ниже, достаточно общим, понятие кортежа в то же время не сопровождается никакими теоретическими построениями; в принципе, им можно пользоваться по основе небольшого числа интуитивно понятных и логически естественных предпосылок.

Кортеж (***tuple***) – это конструкция, представляющая собой простое перечисление нескольких сущностей. В простейшем случае это могут быть значения (выражения), вообще говоря, различных типов. Кортеж строится путём явного перечисления элементов, разделяемых запятыми, и заключённого в круглые скобки, например:

```
(1, 2.35, "string", true)
(x-y, x+y)
(T1(), T2(1), T3(), T1(1))  // здесь T1, T2, T3 - типы
```

Вообще говоря, кортеж может включать как значения, так и другие сущности, например, *типы*. Это утверждение кажется несколько парадоксальным, однако, как мы увидим далее, кортежи, содержащие типы, бывают полезны в некоторых распространённых случаях.

Но сначала обсудим, что можно делать с кортежами. Прежде всего, возникает вопрос, как объявлять объект-кортеж и как обращаться к элементам кортежа. Соответствующие конструкции достаточно очевидны и могут быть проиллюстрированы следующими примерами:

```
z is (1, 2.35, "string", true)
            // объявление объекта-кортежа
x is z(1)   // обращение к первому элементу кортежа
y is z(2)   // обращение ко второму элементу
```

Как видно из примера, синтаксис обращения к элементам кортежа традиционен и образуется из имени объекта-кортежа и порядкового номера элемента, заключённого в круглые скобки. Нумерация элементов кортежа всегда начинается с единицы.

Важно сделать одно замечание. Так как кортеж в общем случае – объединение элементов различных типов, то при компиляции программы тип конструкции вида *кортеж*(*выражение*) в общем случае неизвестен и не может быть определён (единственное, что можно сказать о таком объекте, - это его принадлежность к

типу Any). Такие конструкции допускаются в языке, однако для работы с элементом кортежа, полученным таким образом, требуется последующая динамическая идентификация типа, например:

```
t is (1, true, “string”)
if t(1) is Integer then
   t(1) := t(1) + 1
end
if t(2) is Boolean then
   t(2) := not t(2)
end
```

Если же выражение в конструкции доступа представляет собой константное выражение, то её тип может быть определён при компиляции и тем самым не потребует поддержки во время выполнения:

```
t(2)    // имеет тип Boolean
t(3)    // имеет тип String
```

Как следует из примера, тип кортежа определяется компилятором автоматически. Так, тип объекта info в объявлении вида:

```
info is (x, y)
```

компилятор трактует как (T1, T2), если значения x и y имеют, соответственно, типы T1 и T2. В то же время, возможен, как обычно, полный вариант объявления:

```
info: (T1, T2) is (x, y)
```

Легко заметить, что кортеж можно рассматривать как **обобщение понятия массива** для случая, когда элементы имеют различные типы. Или, что то же самое, **массив – это частный случай кортежа**, когда все его элементы имеют один и тот же статический тип. Отметим также, что доступ к элементам массива и элементам кортежа синтаксически неотличимы и семантически одинаковы. По этой причине нет смысла вводить «разделительную линию» между этими двумя понятиями. По существу, это одна и та же структура данных. Таким образом, тип объекта из следующего объявления

```
odds is (1, 3, 5, 7)
```

можно считать либо массивом, либо кортежем, без изменения семантики.

Помимо естественного обобщения массивов, кортежи обладают рядом других интересных и полезных свойств. Первое такое свойство заключается в возможности снабжать все или некоторые элементы кортежа *именами*. Вот как могут выглядеть простые кортежи с поименованными элементами:

```
t is (first:77, second:1.234, third:true, forth:"string")
smith_info is
   (name:"Smith", emplCode:117766, age:33, salary:122.12)
```

Если элементы кортежа имеют имена, то обращение к элементам может быть задано либо посредством стандартной точечной записи вида Имя-кортежа.Имя-элемента, либо обычной позиционной записью вида Имя-кортежа(Выражение), в которой после имени кортежа в скобках указывается порядковый номер элемента.

Примеры:

```
t.first := 99     // Две данные формы
t(1) := 99        // эквивалентны
```

```
smith_info.salary += 100
smith_info(3)++
```

Наиболее очевидным (и на практике, возможно, самым распространённым и полезным) случаем использования кортежей служит необходимость возврата функцией нескольких значений. Для этого достаточно сформировать кортеж, перечислив типы возвращаемых значений:

```
movePoint(x, y, a: Real): (Real, Real) is
    x := x + a
    y := y + a
    return (x, y)
end
```

Или, в более компактной записи, задав результат непосредственно:

```
movePoint(x, y, a: Real) => (x+a, y+a)
```

Другой пример использования кортежа для возврата из функции двух действительных корней квадратного уравнения можно увидеть в разделе, посвящённом пред- и постусловиям.

Таким образом, в простейшем варианте кортеж – это просто несколько (возможно, поименованных) значений, объединённых в единую конструкцию. Однако, в общем случае ситуация несколько сложнее. Так, в примере функции `movePoint`, приведённом выше, мы задали в качестве типа возвращаемого значения список типов-компонентов, заключённый в скобки: `(Real, Real)`. Эта конструкция тоже считается кортежем, так как она, по существу, описывает реальные значения-кортежи, возвращаемые функцией.

Далее, такая широко используемая конструкция, как *список формальных параметров подпрограмм*, также может считаться кортежем. В отличие от простого списка значений или типов, как в предыдущих примерах, кортеж, представляющий параметры подпрограмм, включает, вообще говоря, полные спецификация компонентов, включая их имена, типы и, возможно, значения по умолчанию.

Наконец, *список фактических параметров* в вызовах подпрограмм тоже считается кортежем. Такая трактовка имеет и некое формально-логическое обоснование, согласно которому семантику операции вызова можно понимать как *применение* заданного кортежа к подпрограмме, в результате которого тело подпрограммы выполняется для значений формальных параметров из кортежа с фактическими параметрами.

Подробнее о параметрах подпрограмм см. главу 11 далее.

> Необходимо пояснить, что последние рассуждения носят несколько умозрительный характер в том смысле, что они, по сути, показывают отличный от традиционного взгляд на привычные языковые конструкции. Эти рассуждения не влияют на семантику подпрограмм; они, скорее, иллюстрируют общность понятия кортежа.

Следует отметить, что синтаксис кортежей в общем случае довольно громоздок. Это не составляет проблемы, если кортежи используются для явного задания значений; однако в случае, когда несколько объектов должны иметь один и тот же тип, который задается в виде кортежа, то текстуальное повторение такого кортежа может привести к загромождению программы. Что даже более существенно, это то обстоятельство, что текстуальное повторение не обеспечивает эквивалентность типов для переменных. Так, две переменные, объявленные, например, следующим образом:

```
a1: (Integer, Real, Boolean)
a2: (Integer, Real, Boolean)
```

по определению имеют *различные* (хотя и структурно совпадающие) типы, и потому не могут использовать совместно:

```
a1 := a2   // ошибка: рассогласование типов
```

Подробнее об именной эквивалентности типов см. главу 12.

> Проблема: принимать ли в языке правило именной эквивалентности типов? Есть мнение, что структурная/именная эквивалентность – это устаревшие понятия, которые правильнее было бы заменить на более универсальное правило *конформности*. Применительно к приведенному примеру, это правило может быть сформулировано так:
>
> Два кортежа считаются конформными, если типы их элементов попарно конформны. Если длины двух кортежей не равны друг другу, то считается, что более длинный кортеж конформен короткому.

Чтобы обеспечить эквивалентность типов для различных переменных-кортежей, следует либо задать их тип в пределах одного объявления:

```
a1, a2: (Integer, Real, Boolean)
```

либо, если их объявления текстуально разнесены, воспользоваться механизмом задания типов «по образцу», которая описана в той же главе 12. Так, если переменные `a1` и `a2` описаны следующим образом

```
a1: (Integer, Real, Boolean)
...
a2: as a1
```

то они по определению будут иметь один и тот же тип.

11. Подпрограммы

Подпрограмма (***routine***) – общее название для структурных единиц программы, которые заключают некоторую последовательность действий, направленных на вычисление нового значения (подпрограмма-***функция***, function routine или просто ***function***), либо на изменение состояния программы (подпрограмма-***процедура***, procedure routine или просто ***procedure***). Функции-***операции*** (operator functions или просто ***operators***) используются как альтернатива функциям для поддержки традиционной (инфиксной или префиксной) нотации записи выражений. Кроме того, в понятие подпрограмм входят ***инициализаторы*** (***initializers***), предназначенные для установки начального состояния экземпляров объектов при их создании.

Подпрограммы могут выступать как в роли программных единиц (компонентов программы самого высокого уровня), так и входить в качестве составных частей в контейнеры, другие подпрограммы или блоки. Подпрограмма самого внешнего уровня вложенности иногда называется отдельно-стоящей подпрограммой (***standalone routine***). Подпрограммы, входящие в состав контейнера, называются ***методами*** (***methods***) или ***свойствами*** (***features***) контейнера. Подпрограмма может быть ***вложена*** в другую подпрограмму (***nested routine***). Наконец, подпрограмма из некоторого блока называется ***локальной*** (***local***) по отношению к этому блоку, аналогично локальным объектам этого блока.

Инициализаторы могут объявляться только в составе некоторого контейнера и служат для установки начального состояния контейнера и объектов соответствующего типа.

```
Подпрограмма
      : Идентификатор-подпрограммы
            [Formal-Generics] [Объявление-параметров] [ : Тип ]
            [Директива-использования]
            [Предусловие]
        Тело-подпрограммы

Идентификатор-подпрограммы
      : [ pure | safe ] [ override ]
                  (Идентификатор | Знак-операции)
      | init

Тело-подпрограммы
      : is [ Объявления-и-операторы | none ] [Постусловие]
                                                end [Завершение]
      | => Выражение
      | is abstract
      | is external

Завершение
      : Идентификатор | Знак-операции | init
```

С синтаксической точки зрения, все подпрограммы устроены максимально похоже. Различение процедур, функций и операций производится по их «вторичным признакам». Так, в объявлении функции и операции обязательно наличие типа возвращаемого значения (или специального синтаксиса, непосредственно задающего возвращаемое значение), которое отсутствует у процедур. Процедуры и обычные функции имеют имена (идентификаторы или составные имена), в то время как функции-операции именуются знаками операций, которые они реализуют. Инициализаторы идентифицируются специальным служебным словом `init`.

Тело подпрограммы состоит из последовательности операторов и/или объявлений, которые в совокупности реализуют семантику действий, выполняемых подпрограммой. Эта последовательность завершается служебным словом **end**.

Подпрограмма-член абстрактного контейнера может быть объявлена как абстрактная (см. раздел XXX). В этом случае телом подпрограммы служит служебное слово **abstract**, в предположении, что в производных контейнерах переопределяемая подпрограмма с тем же именем будет объявлена полностью. Пример:

```
// Подпрограмма в базовом контейнере
move() is abstract

// Подпрограмма в производном контейнере
override move() is /* Тело подпрограммы */ end move
```

Если алгоритм подпрограммы реализован отдельно от её спецификации (например, подпрограмма написана на другом ЯП и/или доступна только в виде выполняемого кода, быть может, в составе некоторой другой программной системы или библиотеки), то в качестве тела подпрограммы задаётся служебное слово **external**, например:

```
Sqrt(x: Real): Real require x>=0 is external
```

Конкретное расположение внешней подпрограммы в файловой системе, её статус (доступность исходного кода, используемое соглашение о вызовах) и другая информация, необходимая для её подключения, задаются отдельно от исходного текста.

> Считается, что конкретное местонахождение внешней реализации подпрограммы (например, файловый адрес её объектного кода) является аспектом окружения программы, будет задаваться в конфигурационном файле и использоваться при сборке программной системы. В самой программе отсутствуют указания на какие-либо технические подробности окружения.

~~Объявление внешних подпрограмм допускается только для подпрограмм самого внешнего уровня вложенности, то есть, для программных единиц.~~

Как видно из синтаксических правил, последовательность объявлений и/или операторов тела подпрограммы завершается служебным словом **end**. ~~При необходимости после **end** можно указать имя подпрограммы.~~

> Повторное задание имени подпрограммы бывает полезным для повышения читабельности программ. Практика показывает, что в случае достаточно сложного алгоритма подпрограммы с обилием вложенных операторов в конце тела подпрограммы сосредотачивается большое число **end**'ов. Возможность маркировать уникальным именем завершающий **end** подпрограммы способствует повышению наглядности. См. также раздел **Метки операторов**.

Тело подпрограммы может быть пустым. Предполагается, что такая возможность будет полезна в процессе разработки, когда необходимо только обозначить наличие подпрограммы без детализации её алгоритма. Компилятор будет автоматически генерировать для таких подпрограмм тело, состоящее из единственного оператора возврата, и может выдавать соответствующее диагностическое сообщение (предупреждение).

> В качестве альтернативы рассматривался более простой синтаксис, когда отсутствующее тело подпрограммы представляется единственным словом **end**, например:

```
P(i:Integer) is
end P
```

Представляется, однако, что подобный стиль программирования не следует поощрять. Более аккуратным решением для обозначения пустого тела было бы явно подчеркнуть неслучайность объявления пустой подпрограммы посредством специального служебного слова. Так, вместо пустого тела стилистически более правильным было бы написать эквивалентное объявление

```
P(i:Integer) is
    none
end P
```

В настоящее время в языке оставлено оба варианта. Выбор конкретной нотации оставляется на усмотрение программиста или правил, принятых в той или иной организации или коллективе разработчиков.

Параметры подпрограмм

Преимущество подпрограмм по сравнению с обычными блоками кода заключается в возможности их параметризации. Тем самым, тело подпрограммы может быть многократно активировано для различных наборов фактических параметров (аргументов).

Параметры подпрограмм задаются в их объявлении после имени подпрограммы (для случая операций – после знака операции, а для инициализаторов – после служебного слова `init`).

```
Объявление-параметров
    : ( [ Секция-параметров { (,|;) Секция-параметров } ] )

Секция-параметров
    : Объявление-переменных
```

В качестве разделителя секций параметров можно использовать как запятые, так и точки с запятыми. Если подпрограмма не имеет параметров, то пустые круглые скобки можно опустить.

Синтаксис объявлений параметров идентичен объявлению обычных переменных. Инициализация параметра (выражение из фразы `is Выражение`), если она имеется, используется как значение параметра по умолчанию. Если для некоторого параметра задано значение по умолчанию, то все последующие параметры также должны иметь значения по умолчанию. Пример:

```
func(x: Real; y is 1, z is 0): Real is ...
```

Если в подпрограмме имеются параметры со значениями по умолчанию, то соответствующие аргументы в вызове этой подпрограммы могут быть опущены. Если в вызове некоторый аргумент отсутствует, то все последующие аргументы также должны отсутствовать. Опущенные в вызове аргументы принимают значения, заданные по умолчанию в объявлении подпрограммы. Например, два следующих вызова эквивалентны:

```
func(11.5)
func(11.5, 1, 0)
```

Заметим, что для однозначной идентификации подпрограммы достаточно информации о типе её формальных параметров. Поэтому, в отличие от обычных объявлений, в объявлениях параметров подпрограмм можно задавать только тип. Пример:

```
sin(Real): Real is external
```

Возможность опускать имена параметров может облегчать восприятие сигнатур подпрограмм в случае отсутствия в объявлении тела подпрограммы, то есть, для абстрактных или внешних подпрограмм. Кроме того, возможно использование «фиктивных» параметров, не используемых в теле подпрограммы для различения нескольких совместно используемых подпрограмм. Наконец, при задании функциональных объектов имена параметров могут быть просто несущественны.

Подпрограммы с переменным числом параметров

Основные положения (*уточнить и снабдить примерами*).

- В языке отсутствуют специальные правила, задающие соответствие числа формальных и фактических параметров. Аналогично, отсутствуют какие-либо синтаксические конструкции, задающие «произвольное» число фактических параметров.

- Любая подпрограмма может быть вызвана с любым числом аргументов (фактических параметров), б***о***льшим или равным числу формальных параметров в объявлении подпрограммы.

- Если число аргументов в вызове больше числа формальных параметров, то «лишние» аргументы в общем случае не рассматриваются и непосредственно (по именам) недоступны.

- Формально считается, что при вызове подпрограммы ей передаётся *кортеж*, элементы которого представляют собой выражения-аргументы. Каждый элемент такого кортежа поименован именем, совпадающим с именем соответствующего формального параметра. Этот кортеж доступен внутри подпрограммы посредством обращения к функции из стандартной библиотеки. Таким образом, «лишние» аргументы всегда доступны посредством обычных обращений к элементам кортежа.

- Если число аргументов в вызове меньше числа формальных параметров, то вызов считается корректным, если недостающие аргументы в теле подпрограммы не используются (при этом возможно предупреждение компилятора). Обращение к параметру, для которого не указан аргумент, считается ошибкой компиляции.

<A вот этого мы не обсуждали совсем – я против такого. В зависимости от тела вызов может быть правильным или нет – по-моему это крамола. А ведь ты понимаешь, что надо учитывать все возможные тела с учетом переопределений – так что предлагаю это убрать!!!>

Предусловия и постусловия

Пред- и постусловия являются важнейшими элементами подхода к программированию, получившего название «контрактное проектирование» (design by contract) и который способствует разработке надежного и безопасного ПО. Основная идея контрактного программирования достаточно проста и заключается в задании необходимых условий корректного начала и завершения работы подпрограмм.

Конкретно, в теле подпрограммы могут быть заданы выражения булевского типа, истинность которых будет автоматически проверяться, соответственно, при каждом вызове подпрограммы и непосредственно перед возвратом управления в контекст вызова.

В типичном случае предусловие содержит проверки смысловой корректности значений параметров подпрограммы и/или корректности их сочетаний. Постусловие включает проверку правильности проведенных подпрограммой действий перед возвратом управления. Принципиальным аспектом контрактного

подхода служит проверка корректности предусловия до начала каждого выполнения подпрограммы и невозможность выполнения её алгоритма в случае нарушения предусловия, а также отказ от возврата управления из подпрограммы в контекст вызова в случае нарушения постусловия (такой отказ может выражаться в возбуждении исключительной ситуации).

```
Предусловие
      : require [else] Список-предикатов

Постусловие
      : ensure [then] Список-предикатов

Список-предикатов
      : Предикат { Разделитель Предикат }

Предикат
      : [ Метка-предиката : ] Выражение
```

Формально пред- и постусловия считаются компонентами спецификации подпрограммы, наряду со спецификациями её формальных параметров и типа возвращаемого значения.

Предусловие задается как последовательность выражений булевского типа, начинающихся со служебного слова **require**. Постусловие строится аналогично, но начинается со служебного слова **ensure**. Пред- или постусловие считается удовлетворённым, если все выражения, входящие в его состав, вырабатывают истинное значение.

> Может показаться, что пред- и постусловия семантически эквивалентны имеющимся во многих языках программирования функциям `assert`, `check` и т.п. Такие функции также проверяют истинность заданных выражений и в случае неудачной проверки реагируют определённым образом. В некотором отношении это действительно так, однако, имеется фундаментальное отличие подобных функций от обсуждаемых конструкций. Это отличие заключается в том, что пред- и постусловия поддерживаются на уровне самого языка. Иными словами, компилятор или другая языковая утилита (статический анализатор, например) будет в состоянии проверить корректность условий на этапе анализа и компиляции программы и, возможно, сделать определённые выводы относительно корректности программы.

Каждое выражение в пред- и постусловии может быть снабжено меткой, которая играет роль комментария, обозначая смысл данного условия. Кроме того, такая метка будет выведена в виде части диагностического сообщения в случае нарушения условия, способствуя тем самым большей информативности сообщения.

Примеры:

```
// Вычисление корней квадратного уравнения,
// заданного тремя коэффициентами.
Roots(a: Real, b: Real, c: Real) : (Real, Real) is
  require
     Главный_коэффициент_ненулевой: a /= 0
     Дискриминант_неотрицательный: b*b - 4*a*c >= 0
  d is Math.sqrt(b*b - 4*a*c)
  return ((-b+d)/(2*a),(-b-d)/(2*a))
end Roots

P is
    // В теле подпрограммы производятся некоторые
    // сложные вычисления с участием переменной
    // из внешнего контекста, причём эта переменная
```

```
        // не должна изменяться.
        ensure
           Глобальный_контекст_не_изменяется:
                          someGlobal = old someGlobal
    end P
```

В постусловии последнего примера используется выражение вида **old** Имя-переменной. Такая конструкция по определению вырабатывает то значение переменной, заданной в выражении, которое эта переменная имела до начала выполнения подпрограммы.

В ряде случаев в постусловии бывает необходимо задать значение, возвращаемое функцией, например, проверить это значение на выполнение того или иного необходимого условия. Ссылка на результат, возвращаемый функцией, внутри постусловия представляется тем же служебным **return**, который используется в операторе возврата:

```
    func(x: Real; y: Integer): Integer is
        // Сложные вычисления...
        return Результат
        ensure
           return in -1..1
    end func
```

Подчеркнем, что служебное слово **return** может использоваться как обозначение возвращаемого значения только внутри постусловия.

В случае, когда выражение в предусловии вырабатывает значение **false**, вызов подпрограммы не происходит, а возбуждается исключительная ситуация RequireViolation. Если выражение в постусловии вырабатывает значение **false**, возврат управления в точку вызова не происходит, а возбуждается исключительная ситуация EnsureViolation. <***Подробнее***>

<Смысл «уточнённых» пред- и постусловий **require else** и **ensure then**:

При наследовании в случае переопределения подпрограммы предусловия расширяются, а постусловия заужаются – именно это и отражается формами require else & ensure then.

Функции

Функцию можно считать абстракцией для вычисления значения. Тип значения, вырабатываемого функцией, задаётся после списка её параметров. Так как функция возвращает значение, её вызов трактуется как выражение (точнее, первичное-выражение, см. раздел **Выражения**) и тем самым может входить в другие выражения в качестве операнда некоторой операции.

Пример:

```
    product(a, b, c: Real): Real
    is
       return sqrt(sqr(a)+sqr(b)+sqr(c))
    end product

    // Использование вызова функции в позиции выражения
    p: Real is product(1.2, 3.4, 5.6)
```

В некоторых случаях, типичным примером которых служит функциональный стиль программирования, функции играют важную роль, выступая не только как абстракции вычисления значений, но и как самостоятельные объекты, которые можно присваивать, передавать параметрами в другие функции и возвращать в качестве результата выполнения других функций. Парадигма функционального

подхода предполагает, что такие функции, как правило, производят простые вычисления, которые могут представляться единственным выражением.

В целях задания более компактной и наглядной нотации для подобных случаев допускается сокращённая форма объявления функции. Так, если тело функции содержит единственный оператор вычисления возвращаемого значения (примером служит предыдущее объявление функции), то объявление такой функции можно записать в более компактном виде:

```
product(a, b, c: Real): Real => sqrt(sqr(a)+sqr(b)+sqr(c))
```

Более того, в этом случае компилятор может вывести тип значения, возвращаемого функцией, из вида самого выражения. Поэтому явное задание типа в данном случае можно опустить:

```
product(a, b, c: Real) => sqrt(sqr(a)+sqr(b)+sqr(c))
```

Приведём еще несколько примеров сокращённой записи функций:

```
Sum(a, b: Real) => a+b
Max[T extend Comparable](a, b: T) => if a>b then a else b
Что такое extend Comparable?
```

Чистые и безопасные функции

Выполнение процедуры (её вызов) заключается в наступлении некоторого *побочного эффекта*, то есть, изменения значений внешних по отношению к ней объектов программы и, шире, в изменении состояния программы. В отличие от этого, основной смысл выполнения функций состоит в вычислении некоторого нового значения.

С другой стороны, любая функция, помимо возврата значения, вообще говоря, тоже может производить некоторый побочный эффект. Язык SLang, как и большинство современных языков, не содержит никаких ограничений на этот счёт. С другой стороны, в очень многих случаях подобная двойственная природа функций представляется нежелательной или недопустимой. В частности, парадигма функционального программирования в принципе не предполагает возникновения побочных эффектов в результате вызовов функций. Кроме того, функции, гарантированно не вызывающие побочных эффектов, имеют ряд преимуществ: в частности, они оставляют достаточно широкие возможности оптимизации, а также допускают эффективное распараллеливание вычислений.

По указанным причинам, в языке имеется дополнительная возможность задания так называемых «чистых» (pure) функций. Если функция объявлена как чистая, то операторы её тела могут использовать исключительно значения фактических параметров, переданных ей при вызове. Объявление такой функции должно включать описатель **pure** перед её именем, например:

```
// Функция sin гарантированно не производит никаких
// побочных эффектов и не использует никакой внешней
// информации кроме значений аргументов
pure sin(x: Real): Real is ... end
```

Должно быть достаточно понятно, что в ряде случаев компилятор в состоянии самостоятельно детектировать «чистоту» функции, проанализировав операторы её тела. По этой причине описатель **pure** можно считать своего рода подсказкой компилятору. В то же время предполагается, что явное задание чистых функций будет способствовать повышению наглядности программ, ясно информируя читателя об особенностях программы.

Заметим, однако, что в ситуации, когда тело функции отсутствует (например, для случая внешних подпрограмм), задание спецификатора **pure** имеет характер необходимости, так как в этом случае компилятор не сможет самостоятельно сделать вывод о «чистоте» функции:

```
// Функция sin гарантированно не производит никаких
// побочных эффектов. Компилятор не сможет удостовериться
// в этом, так как исходный текст тела функции недоступен.
// Поэтому спецификатор pure необходим.
pure sin(x: Real): Real is external
```

Спецификатор **safe** предназначен для аналогичных целей, но в более «мягком» варианте. А именно, задание для функции данного спецификатора информирует, что эта функция использует (по чтению) значения объектов, внешних по отношению к ней. В число таких внешних объектов могут, в частности, входить атрибуты контейнера, в котором объявлена эта функция.

```
globalDelta: Integer

unit A
    value: Integer
    safe shift => value + globalDelta
end // A
```

Совместное использование подпрограмм

<Могут возникнуть проблемы совмещения совместного использования (overloading) и принципа произвольного числа аргументов…>

<Проблем нет так как мы проверяем правильность каждого обращения >

Функции-операции

Функции-операции могут рассматриваться как вариант функций, предназначенных для задания реализаций традиционных инфиксных и префиксных операций для операндов типов, задаваемых контейнерами.

Именем функции-операции служит знак переопределяемой операции из числа допускающих переопределение. Функция-операция всегда является частью некоторого контейнера и определяет некоторую операцию для объектов типа, задаваемого этим контейнером.

Число параметров функции-операции должно строго соответствовать числу операндов операции, знак которой переопределяется. Именно, функция-операция имеет ноль параметров, если она переопределяет унарную операцию, или один параметр для случая переопределения бинарной операции.

> Причина введения такого правила должна быть достаточно понятна: все функции-операции, как и любые подпрограммы в контейнерах, имеют дополнительный неявный параметр, тип которого совпадает с типом контейнера, в котором они объявляются.

В отличие от обычных функций, средством активации которых является конструкция «вызов», операции могут использоваться в инфиксной форме в составе выражений – по тем же правилам, что и стандартные инфиксные операции для библиотечных типов.

На пользовательские версии операций накладываются следующие ограничения:

1. В качестве переопределяемых знаков операций могут выступать знаки (лексемы) из набора знаков операций, определенных в языке (то есть, соответствовать синтаксическому правилу Знак-операции). Иными словами, собственные знаки операций определять невозможно.

> Идея такого ограничения была сформулирована автором языка C++ Б.Страуструпом: «Язык должен быть расширяемым, но не изменяемым».

2. Приоритет переопределяемой операции и её арность (число операндов) должны соответствовать приоритету и арности операции с таким же знаком, определённой для библиотечных типов (см. раздел «**Стандартная библиотека**»).

Пример. Вот как может выглядеть

```
unit Rational extend Numeric

    numer, denom: Integer

    init(n, d: Integer) is
        require d /= 0
        g is gcd(n.abs, d.abs)
        numer := n / g
        denom := d / g
        gcd(a, b: Integer): Integer =>
           if b = 0 then a else gcd(b, a % b)
    end

    init(n: Integer) is init(n, 1) end

    + (that: Rational): as Rational =>
        Rational(numer * that.denom + that.numer * denom,
                 denom * that.denom)

    + (i: Integer): as Rational => Rational(numer+i*denom, denom)

    - (that: Rational): as Rational =>
        Rational(numer * that.denom - that.numer * denom,
        denom * that.denom)

    - (i: Integer): as Rational => Rational(numer-i*denom, denom)

    * (that: Rational): as Rational =>
        Rational(numer * that.numer, denom * that.denom)

    * (i: Int): as Rational => Rational(numer * i, denom)

    / (that: Rational): as Rational =>
        Rational(numer * that.denom, denom * that.numer)

    / (i: Integer): as Rational => Rational(numer, denom * i)

    override toString => numer.toString +"/"+ denom.toString

end Rational
```

Обобщенные подпрограммы

Оператор возврата: за и против

12. Объекты и их объявления

Общее правило, принятое в языке, заключается в том, что все объекты программы должны быть объявлены. Именно объявление задаёт фундаментальные атрибуты – имя, тип и начальное значение – связывая их в одну сущность и определяя тем самым объект как таковой.

> Заметим, что необходимость объявлять объект не обязательно означает текстуальное следование вида «сначала объявление – потом использование»: это правило действует не во всех контекстах. Здесь только подчеркивается необходимость явного объявления для любого объекта. Правила текстуального следования различаются для различных ситуаций и контекстов и объясняются в соответствующих разделах.

Начнем с самых простых примеров. Вот первый:

```
x is 5
```

Это весьма компактное, но вместе с тем полноценное объявление, 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***, 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: оно задает тип объекта и, тем самым, определяет допустимые контексты его использования. С другой стороны, число `5` становится значением («начальным значением») объекта `x`.

> 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. 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 задания завершителей приводится в разделе **Использование точек с запятой**.

Рассмотрим следующий пример объявления:

```
x is Integer
```

Чтобы глубже понять семантику объявлений, неодолимо прояснить два аспекта: во-первых, какое значение принимает объект `x` из этого объявления, и, во-вторых, почему здесь после слова `is` задается не значение, как в предыдущем примере, а тип?

На самом деле, существует единый ответ на оба поставленных вопроса, и он заключается в следующем: после служебного слова `is` в объявлении *всегда задается выражение*, определяющее значение объявляемого объекта. Иными словами, любой объект получает при объявлении некоторое значение.

Слово `Integer` в последнем примере обозначает, строго говоря, не целочисленный тип вообще, а задает ***инициализатор***, определенный для типа `Integer`. Полная форма вызова инициализатор выглядит как `Integer()`, или, более полно, `Integer.init()`, однако пустые круглые скобки и явное указание инициализатора можно опускать.

Инициализатор типа по определению вырабатывает новый объект данного типа, и этот объект становится значением объявляемого объекта.

Возвращаясь к первому примеру, становится ясно, что запись вида

```
x is 5
```

является более короткой формой объявлений

```
x is Integer(5)
x is Integer.init(5)
```

или

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

При необходимости, тип объекта можно задать явно. Если начальное значение объекта в объявлении не задается, такое явное задание является обязательным:

```
x: Integer
```

В этом случае начальным значением переменной `x` становится значение, вырабатываемое инициализатором типа `Integer` без параметров.

Из предыдущих рассуждений ясно, что все приведённые примеры фактически являются сокращениями. Полный вариант приведенных примеров объявлений выглядит следующим образом:

```
x: Integer is Integer.init(5)
```

или даже

```
x: Integer is new Integer.init(5)
```

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

```
odds is (1, 3, 5, 7)
```

Конструкция после служебного слова `is` представляет собой массив. Эта запись содержит всю необходимую информацию об объявляемом объекте: скобки служат признаком массива, элементы, заключенные в скобки (в общем случае это могут быть произвольные выражения), становятся элементами массива; их количество задает размер массива, а тип выражений (в данном примере он один и тем же для каждого элемента списка) определяет тип элементов массива.

Таким образом, объект `odds` из объявления выше получает тип, который может быть явно записан как

```
Array[Integer]
```

Как видно из формы задания массивов, библиотечный тип `Array` представляет собой обобщенный («generic») класс, типовой параметр которого задает тип элементов.

При необходимости можно явно указать тип объявляемого объекта. Эквивалентная форма записи выглядит так:

```
odds: Array[Integer] is (1, 3, 5, 7)
```

Если инициализирующее выражение полностью определяет свой тип, то компилятор может автоматически вывести этот тип; тем самым, допускается неполное задания типа после двоеточия, например, `Array` вместо `Array[Integer]`. Ниже показано несколько семантически эквивалентных вариантов одного и того же объявления:

```
odds: Array is (1, 3, 5, 7)
odds: Array is Array(1, 3, 5, 7)
odds: Array[Integer] is (1, 3, 5, 7)
odds: Array is Array(1, 3, 5, 7)
odds: Array[Integer] is Array(1, 3, 5, 7)
odds: Array[Integer] is Array[Integer].init(1, 3, 5, 7)
```

> Заметим, что возможности автоматического выведения типа компилятором зависят от конкретного типа. Список альтернативных объявлений, представленный выше для массива, может быть другим для некоторых других типов.

Объявление вида

```
t is (123, true, 456, false)
```

вводит в текущий контекст программы объект с именем `t`, тип которого, как и в предыдущих примерах, выводится компилятором из типа инициализирующего выражения. В данном случае типом `t` является кортеж из четырёх элементов, первый и третий из которых имеет тип `Integer`, а второй и четвёртый – тип `Boolean`.

Формальный синтаксис объявлений описывается следующими правилами:

```
Объявление-объектов
     : [ const ] Объявление-переменных

Объявление-переменных
     : Список-идентификаторов [ : Спецификатор-типа ]
                                          is Выражение
     | Список-идентификаторов : Спецификатор-типа

Спецификатор-типа
     : [ ? | ref | val ] Тип
     | as Составное-имя
```

Как следует из приведенных правил, в объявлении можно задать несколько имен. В этом случае они отделяются друг от друга запятыми, например:

```
a, b, c is 77
```

Инициализирующее выражение в таком объявлении вычисляется *один раз*, и полученное значение становится начальным значением *для всех* объявляемых переменных.

Префикс **`const`** перед объявлением обозначает объявление константы – объекта, значение которого не может изменяться. В частности, константа не может появиться в левой части оператора присваивания. Если в объявлении задан префикс **`const`**, то задание начального значения является обязательным.

Задание типа по образцу. Эквивалентность типов

В ряде случаев бывает необходимо назначить один и тот же тип нескольким переменным, возможно, находящимся текстуально далеко друг от друга (возможно, в различных программных единицах). Если тип представляется достаточно громоздкой записью, то удобно при его повторном задании указать ссылку на тип, ранее назначенный некоторой другой переменной, не выписывая этот тип ещё раз.

Вот пример:

```
unit M1
    a: Dictionary[Integer, String]
    b: (first:Integer; second:Boolean)
    ...
end M1

unit M2
    x: as M1.a // то же, что x: Dictionary[Integer, String]
    const y: as M1.b is (1, false)
end M2
```

В этом примере переменная `x` получает в точности тот же тип, что и переменная `a` из контейнера `M1`, а константа `y` – тот же тип, что и переменная `b` из того же контейнера.

> Необходимо решить, можно ли посредством **as** ссылаться на объект из другой программной единицы.

Необходимо сделать небольшое замечание. Использование типа по образцу в нашем примере подразумевает, что объекты `M1.a` и `M2.x`, а также объекты `M1.b` и `M2.y` попарно имеют в *точности один и тот же тип*. Однако, если бы во втором контейнере мы текстуально повторили задание типов из первого контейнера, например:

```
unit M2
    x: Dictionary[Integer, String]
    const y: (first:Integer; second:Boolean) is (1, false)
end M2
```

то типы объектов `M1.a` и `M2.x` будут по-прежнему тождественны (**эквивалентны**) друг другу. В то же время вывод об эквивалентности типов `M1.b` и `M2.y` мы бы сделать не смогли. Дело в том, что типы первой пары объектов задаются посредством имени этого типа. В языке SLang принято правило *именной эквивалентности типов*, согласно которому типы объектов эквивалентны в том и только том случае, когда они заданы одним и тем же именем.

В то же время, типами объектов `M1.b` и `M2.y` служат кортежи, которые заданы явно, посредством указания их компонентов. Типы этих объектов не имеют имени, поэтому правило именной эквивалентности для них неприменимо. Таким образом, считается, что типы указанных объектов различны. Это не позволяет, в частности, присваивать их значения друг другу:

```
M1.b := M2.y     // ошибка: рассогласование типов
```

В то же время компоненты этих объектов имеют типы, заданные посредством (одних и тех же) имён, поэтому почленное присваивание допустимо:

```
M1.b.first := M2.y.first       // ok
M1.b.second := M2.y.second     // ok
```

Принцип именной эквивалентности типов на практике обеспечивает наиболее простое и надёжное средство обеспечения корректности программ, поэтому этот

принцип используется практически во всех современных языках программирования.

> Одним из последних универсальных ЯП, в которых использовался принцип структурной эквивалентности типов, был Algol-68.

Возвращаясь к нотации «тип по образцу», отметим, что такая форма записи обеспечивает простую и наглядную спецификацию для задания объектов, типы которых должны быть эквивалентны.

> Весь последний фрагмент следует переписать с заменой идеи именной эквивалентности типов на правило конформности. Во-вторых, прямое присваивание атрибутам, как известно, не допускается ☺.

13. Неинициализированные переменные

Как уже говорилось во введении, язык SLang трактует проблему пустых указателей не как самостоятельную проблему, а как часть более общей проблемы некорректной работы с *неинициализированными переменными*. Конкретно, пустая ссылка считается разновидностью неинициализированной переменной, и в языке имеются механизмы, которые строго ограничивают случаи, когда действительно нужны неинициализированные переменные, от ситуаций, когда всякий объект должен иметь определенное значение. Кроме того, имеются средства перехода от потенциально неинициализированных переменных к инициализированным – своего рода мостик от «опасного» мира в «безопасный».

Основные моменты предлагаемого подхода лучше всего показать на примерах. Рассмотрим библиотечный контейнер `Integer`, который содержит некоторый ассортимент операций над объектами целочисленных типов, а также может определять собственно целочисленный тип. Программное определение такого контейнера имеет следующий вид:

```
val unit Integer
    ...
end
```

Префикс `val` перед объявлением контейнера (см. раздел **Типы-значения и ссылочные типы**) говорит, что по умолчанию все объекты этого типа являются *значениями*, а не ссылками. В то же время объявление вида

```
ri: ref Integer
```

задает *ссылку* на объект типа `Integer`.

Эти предварительные комментарии приведены, чтобы подчеркнуть то обстоятельство, что понятие инициализированности переменной применяется как для ссылок, так и для значений (самих объектов) и, разумеется, не только для типа `Integer`, но для любых типов. Теперь собственно сам пример.

```
a1: Integer is 5
```

Здесь для переменной `a1` задается явная инициализация в форме `is 5`. Заметим, что если бы мы не указали для переменной `a1` явную инициализацию, то ее инициализация была бы выполнена посредством инициализатора без параметров `Integer.init()`, объявленного в контейнере `Integer`.

Семантика языка предполагает, что факт получения атрибутом контейнера некоторого значения после завершения работы процедуры инициализации будет проверен – либо статически, в процессе компиляции, либо системой времени выполнения. Таким образом, общее правило звучит так: «обычное» объявление *всегда* приводит к созданию объекта, имеющего значение.

Наряду с этим, в языке имеется альтернативная форма объявления:

```
a2: ?Integer
```

Атрибут `a2`, объявленный таким образом, *не имеет значения*. Ни компилятор, ни система времени выполнения не будут контролировать факт получения этим атрибутом значения при инициализации. Однако, с таким атрибутом *нельзя выполнять никаких действий*, кроме передачи его в качестве аргумента или присваивания ему значения.

Таким образом, мы можем свободно работать с атрибутом `a1`, присваивая его значение другим атрибутам, объявленным аналогичным образом. Кроме того, допустимым является присваивание `a2 := a1` (в результате этой операции

атрибут a2 получает значение), однако присваивание a1 := a2 считается некорректным; допущение такого действия означало бы неконтролируемое распространение неинициализированного значения.

Если же логика программы требует выполнить подобное присваивание, то для его корректности следует предварительно проверить, имеет ли атрибут a2 в данной точке программы некоторое значение. Отправным моментом такой проверки служит идея, согласно которой *только инициализированный атрибут имеет тип*. Тем самым, требуется узнать, является ли a2 правильным объектом типа Integer. Таким образом, мы сводим проверку на наличие значения *к динамической проверке типа* атрибута.

Вот как выглядит переход от потенциально неинициализированной переменной к безопасной:

```
if a2 is Integer then
    a2 := a2 + 5
    a1 := a2
end
```

Семантика операции is гарантирует, что внутри then-части условного оператора динамический тип атрибута a2 гарантированно имеет тип Integer (или его наследник), и потому обращение к свойствам a2 или присваивание его другим инициализированным атрибутам является заведомо корректным. Однако, после завершения условного оператора атрибут a2 уже не считается гарантированно инициализированным, и опять действуют ограничения на работу с ним.

В дополнение к описанному механизму, наряду с операцией получения значения, имеется и «симметричная» операция *потери значения* (или отмены инициализации), и она может быть выполнена над a2 в любой момент выполнения:

```
?a2
```

Такая операция, очевидно, корректна только для атрибутов, объявленных как неинициализированные. Поэтому, например, конструкция ?a1 приведет к ошибке времени компиляции.

> Представляется, что предложенное решение – более полное и практически более удобное, нежели сходные решения в других языках программирования, например, в C#, Kotlin, Eiffel и других.

Раздел завершается примером более практического характера, который показывает использование описанного механизма совместно с типовой параметризацией.

```
unit List[G]
  item is G
  next is ?List[G]
  setItem(other: G)
    item := other
  end
  add(other: G)
    next := List[G](other)
  end

  init(element: G)
    setItem(element)
    ?next
  end
end
```

```
// Создаем список из одного элемента
list1 is List[Integer](5)
// Объявляем пустой список
list2 is ?List[Integer]
list2 := list1

if list2 is List[Integer] then
  // Добавляется еще один элемент
  list2.add(128)
  while list2 is List[Integer] loop
    // Цикл работает, пока очередной list2 имеет значение
    StandardIO.put(" ",l2.item)
    // Перейти к следующему элементу списка
    list2 := list2.next
  end
end
```

14. Операторы

```
Statement
       : Присваивание
       | Локальное-объявление
       | IfCase
       | Цикл
       | Завершение
       | FeatureCallOrCreation
       | Потеря-значения
       | Проверка
       | Возврат
       | Блок-с-контролем
       | Возбуждение-исключения

Завершение
       : break [ Метка ]

Метка : Идентификатор

Потеря-значения
       : ? Идентификатор

Проверка
       : check Список-предикатов end

Возврат
       : return [ Выражение ]

Возбуждение-исключения
       : raise [ Выражение ]

Блок-с-контролем
       : try
            Список-операторов
            Обработчик { Обработчик }
            [ else Список-операторов ]
         end

Обработчик
       : catch ( [ Идентификатор : ] UnitType)
            [StatementsList]

Assignment
        : Writable ":=" Expression

AttributeNamesList
        : [const] Identifier {","[const] Identifier}

LocalAttributeDeclaration
        : AttributeNamesList ([":" Type] is Expression)|([":"
"?" Type] [is Expression])

UnitAttributeDeclaration
        : AttributeNamesList (":" Type [is Expression])|(is
Expression)

Writable: Identifier ["("ExprList")"]{"."Identifier
"("ExprList")"}
```

Присваивание

Вызов подпрограммы

Оператор возврата

Условный оператор

Оператор цикла

15. Выражения

Приложение
Стандартная библиотека языка SLang

Как уже говорилось, для значений каждого предопределённого типа имеется фиксированное множество операций. Как правило, операции задаются в традиционной инфиксной или префиксной форме с использованием знаков операций. Знаки операций представляются в виде традиционных обозначений (`+` для операции сложения, `*` для операции умножения, и т.д.). Знаки некоторых операций обозначаются служебными словами языка (например, **`and`** для операции логического умножения). Для некоторых операций поддерживаются альтернативные обозначения: как в виде знаков, так и в виде служебных слов (например, операция взятия остатка от целочисленного деления может быть записана как в виде знака `%`, так и служебным словом **`rem`**).

Тип Integer

Множество значений целого типа включает все возможные (положительные и отрицательные) значения, непосредственно поддерживаемые аппаратурой целевой платформы. Тем самым, конкретное множество целых чисел в определении языка не фиксируется и задаётся реализацией. Аналогично, внутреннее представление целых чисел (прямой, обратный или дополнительный код) не фиксируется и также определяется реализацией. Некоторые атрибуты целого типа, например, размер представления чисел и его максимальные и минимальные предельные значения, доступны посредством обращения к соответствующим предопределённым функциям. Набор операций и функций, определённых для значений целого типа, представляется ниже в этом разделе.

> Шаг в строну в Паскале еще были дипазоны i : 1.. 10 – если правильно понмю синтакис и в Аде такая штука была. А почему бы не интерпретировать запись i : Integer [1..10] как наследование от class Integer с единственной целью расширить его инвариант 1 <= this and this <= 10. Т.е. по сути это просто сокрашенная запись – ну лень мне наследоваться лишь толкьо для того чтобы поменять инвариант класса … Т.е. по сути я создаю наследника но не именую его отдельным идентификатором … Как тебе такая трактовка?

Целый тип задаётся в объявлении переменной либо явно, посредством указания идентификатора `Integer`, либо неявно. В последнем случае тип выводится компилятором из контекста объявления, которым в данном случае является тип инициализирующего значения.

В языке отсутствует понятие «подтипов» целого типа, таких как «байт», «короткое целое», «длинное целое» и т.д. Аналогично, отсутствует деление на «знаковые» и «беззнаковые» целые типы. Вместо подобного набора различных «целых типов» определён единственный целый тип, который, очевидно, включает в себя все возможные подтипы целого. По мнению авторов, это существенно упрощает общую систему типов языка без всякого ущерба для его выразительности и без значительного ущерба для эффективности программ. Предполагается, что во многих случаях оптимизирующий компилятор сможет выбрать для конкретной целочисленной переменной наиболее эффективное машинное представление.

Наряду с этим, общий механизм обобщенных типов дает возможность задания подобных «подтипов» целого. Тип `Integer` определен в библиотеке как обобщенный контейнер вида

```
unit Integer[N:Integer]...
```

Примеры.

```
x : Integer
y : Integer = 777
z := 0o777
```

В первых двух объявлениях тип переменных `x` и `y` явно определяется как `Integer`. Второе объявление содержит задание начального значения переменной, в то время как переменная из первого объявления получает начальное значение по умолчанию.

Третье объявление не содержит явного задания типа. В этом случае компилятор автоматически выведет тип переменной `z` как `Integer`, основываясь на результате анализа типа инициализирующего выражения.

Для значений целого типа определён стандартный набор арифметических операций: сложение, вычитание, умножение, целочисленное деление (с отбрасыванием остатка), взятие остатка от целочисленного деления, а также возведение в целочисленную степень.

Знак операции	**Альтернативный знак**	**Семантика операции**	**Особые случаи**
`+`		Сложение: два целочисленных операнда алгебраически складываются	Переполнение: возбуждение исключительной ситуации `Overflow`
`-`		Вычитание: значение второго операнда алгебраически вычитается из значения первого операнда	Исчезновение значения: возбуждение исключительной ситуации `Underflow`
`*`		Умножение: значения операндов алгебраически перемножаются	Переполнение: возбуждение исключительной ситуации `Overflow`
`/`	**`mod`**	Целочисленное деление: значение первого операнда алгебраически делится на значение второго операнда. Результатом операции служит целая часть от результата деления	Деление на ноль: возбуждение исключительной ситуации `ZeroDivide`
`%`	**`rem`**	Взятие остатка от целочисленного деления: значение первого операнда алгебраически делится на значение второго операнда. Результатом операции является целочисленный остаток деления	
`**`	`^`	Возведение в целочисленную степень: значение первого операнда возводится в степень, равную значению второго операнда	Переполнение: возбуждение исключительной ситуации `Overflow`

Кроме арифметических операций, в языке определены операции, трактующие целочисленные значения как набор битов, и выполняющие групповые действия над соответствующими парами битов значений.

Знак операции	Альтернатив-ный знак	Семантика операции	Особые случаи
&	and	Побитовая логическая операция «и»	
|	or	Побитовая логическая операция «или»	
^	xor	Побитовая логическая операция «исключающее или»	
~	not	Побитовая унарная логическая операция отрицания	
<<		Сдвиг битового представления влево на число разрядов из второго (целочисленного) операнда	
>>		Сдвиг битового представления вправо на число разрядов из второго (целочисленного) операнда	

Операции сравнения могут принимать операнды целочисленного типа. Результат операций – значение булевского типа (`true` или `false`):

Знак операции	Альтернатив-ный знак	Семантика операции	Особые случаи
<		Меньше	
<=		Меньше или равно	
>		Больше	
>=		Больше или равно	
=		Равно	
/=		Не равно	

Некоторые предопределённые функции для операндов целого типа:

Имя функции	Аргумент	Семантика функции	Примеры и особые случаи

Max	Переменная целого типа	Возвращает максимально возможное значение целого типа (константа)	Функции не зависят от реального значения аргумента и всегда выдают одно и то же значение (константу, определённую реализацией). Допустимые формы вызова:
			`v.Max`
			`v.Min`
			`Integer.Min`
			`Integer.Max`

Min	Переменная целого типа	Возвращает минимально возможное значение целого типа (константа)	
Size	Переменная целого типа	Реальное число байтов, отведённых системой для хранения данного значения (константа)	`v.Size` `12345.Size`
bytes_count			

Надо помнитьт что size – это размер в байтах – если назвать bytes_count или BytesCount – то все понятно …

1.1.1 Вещественный тип

Множество значений вещественного типа включает представимые аппаратурой и/или операционной средой вещественные значения. Конкретный диапазон допустимых значений вещественных чисел в определении языка не фиксируется и задаётся реализацией. Аналогично, внутреннее представление целых чисел не фиксируется и также определяется реализацией. Некоторые атрибуты целого типа, например, размер представления чисел и его максимальные и минимальные предельные значения, доступны посредством обращения к соответствующим предопределённым функциям. Набор операций и функций, определённых для значений целого типа, представляется ниже в этом разделе.

Вещественный тип, подобно целочисленному, задаётся в объявлении объекта либо явно, посредством указания идентификатора `Real`, либо неявно. В последнем случае тип выводится компилятором из контекста объявления, которым в данном случае является тип инициализирующего значения.

Операции:

- Стандартные арифметические операции: `+`, `-`, `*`, `/`.
- Возведение вещественного числа в целочисленную степень `**`
- Шесть стандартных операций сравнения вещественных значений: `<`, `<=`, `>`, `>=`, `=`, `/=`.

Строго говоря, множество целочисленных значений можно считать подмножеством вещественных значений. По этой причине многие операции допускают операнды смешанных типов. Так, один из операндов перечисленных выше операций может иметь целочисленный тип, а другой – вещественный тип. Исключением служит операция возведения в степень, в которой тип второго операнда фиксирован как целочисленный. Общее правило заключается в том, что результат операций имеет вещественный тип, если хотя бы один из операндов имеет вещественный тип. См. также раздел «Преобразования между предопределёнными типами» ниже.

Is there type Double?

1.1.2 Символьный тип

Символьный тип обозначается предопределённым идентификатором `Char`. Значениями символьного типа служат произвольные символы из набора Unicode.

Операции над значениями символьного типа:

- Сложение символьных значений `+`. Результат операции – строка, состоящая из двух символов-операндов.

Над значениями символьного типа допускается ряд преобразований в значения других типов, см. раздел «Преобразования между предопределёнными типами» ниже.

1.1.3 Строковый тип

Строковый тип обозначается предопределённым идентификатором `String`. Значениями строкового типа являются все возможные цепочки символов из набора Unicode произвольной длины.

Операции над значениями строковых типов:

- Сложение (конкатенация) строк `+`. Результат операции – строка, состоящая из символов первого операнда (в исходном порядке), непосредственно после которых располагаются символы (в исходном порядке) из второй строки-операнда.

- Чтение элемента строки `[]`. Операция имеет синтаксис традиционного вида, с использованием квадратных скобок (см. главу «Выражения»). Результат операции имеет символьный тип.

Допускается преобразование строковых значений единичной длины в символьное значение, см. раздел «Преобразования между предопределёнными типами» ниже. Кроме того, для строкового

типа определён большой набор предопределённых операций, представляемых в виде библиотечных функций из стандартной библиотеки языка, см. главу XXX.

1.1.4 Булевский тип

Булевский тип обозначается предопределённым идентификатором `Boolean`. Имеется два значения булевского типа – «истина» и «ложь», которые представляются служебными солвами `true` и `false`, соответственно.

Над значениями булевского типа определены следующие операции:

<table>
<tr><th>Знак операции</th><th>Семантика операции</th><th>Замечания и примеры</th></tr>
<tr><td>and</td><td>Логическое умножение</td><td>
<table>
<tr><th>Операнд1</th><th>Операнд2</th><th>Результат</th></tr>
<tr><td>false</td><td>false</td><td>false</td></tr>
<tr><td>false</td><td>true</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>false</td></tr>
<tr><td>true</td><td>true</td><td>true</td></tr>
</table>
</td></tr>
<tr><td>or</td><td>Логическое сложение</td><td>
<table>
<tr><th>Операнд1</th><th>Операнд2</th><th>Результат</th></tr>
<tr><td>false</td><td>false</td><td>false</td></tr>
<tr><td>false</td><td>true</td><td>true</td></tr>
<tr><td>true</td><td>false</td><td>true</td></tr>
<tr><td>true</td><td>true</td><td>true</td></tr>
</table>
</td></tr>
<tr><td>xor</td><td>Исключающее «или» (отрицание эквивалентности)</td><td>
<table>
<tr><th>Операнд1</th><th>Операнд2</th><th>Результат</th></tr>
<tr><td>false</td><td>false</td><td>false</td></tr>
<tr><td>false</td><td>true</td><td>true</td></tr>
<tr><td>true</td><td>false</td><td>true</td></tr>
<tr><td>true</td><td>true</td><td>false</td></tr>
</table>
</td></tr>
<tr><td>not</td><td>Логическое отрицание</td><td>Унарная операция: для операнда true результатом служит значение false, для значения false – true.</td></tr>
</table>

1.1.5 Преобразования между предопределёнными типами

Значение некоторого типа может быть преобразовано в значение некоторого другого типа. На преобразование типов значений накладываются достаточно жёсткие ограничения, смысл которых заключается в недопущении случайных или намеренных искажений результирующих значений. Как правило, преобразование типа задаётся явно, посредством конструкции вида

```
Результирующий-тип ( Выражение )
```

```
Вот так и хочется сказать ну чем был плох Си там это писалось
(результирующий тип) Выражение – в чем инновация ? ☺
```

Единственный допустимый случай неявного преобразования – это преобразование от целого значения к вещественному. Оно не может привести к потере данных и потому всегда безопасно. Все остальные допустимые преобразования должны быть заданы явно посредством конструкции «преобразование типа».

В таблице ниже охарактеризованы все возможные случаи преобразований между предопределёнными типами.

Исходный тип	Результирующий тип	Допустим?	Комментарий
`Integer`	`Real`	Да; явно или неявно	«Обобщение» целого до вещественного. Всегда возможно. Делается по умолчанию.
`Integer`	`Boolean`	Да; явно	Допустимо посредством явного преобразования. Для любого ненулевого целого результат есть **true**, в противном случае **false**.

`Integer`	`Char`	Нет	
`Integer`	`String`	Нет	
`Real`	`Integer`	Да; явно	Округление вещественного до ближайшего целого. Допустимо посредством явного преобразования.
`Real`	`Boolean`	Нет	
`Real`	`Char`	Нет	
`Real`	`String`	Нет	
`Boolean`	`Integer`	Да; явно	Допустимо посредством явного преобразования. Результат есть `1` для аргумента **`true`** и `0` в противном случае.
`Boolean`	`Real`	Нет	
`Boolean`	`Char`	Нет	
`Boolean`	`String`	Нет	Имеется предопределённая функция `toString`, преобразующая булевское значение в строковое представление: `true.toString` возвращает строку `"true"` `false.toString` возвращает строку `"false"`
`Char`	`Integer`	Да; явно	Допустимо преобразование, которое должно быть задано в явной форме. Результат преобразования – целочисленное значение, представляющее десятичный код символа-аргумента.
`Char`	`Real`	Нет	
`Char`	`Boolean`	Нет	
`Char`	`String`	Да; явно	Допустимо посредством явного преобразования. Результат преобразования строка длиной 1, единственным символом которой служит символ из аргумента.
`String`	`Integer`	Нет	Ни явное, ни неявное преобразования типа `String` невозможны. Однако, для типа `String` имеются функции `parseInteger`, `parseReal` и `parseBoolean`, которые осуществляют преобразования строковых представлений соответствующих типов в значения этих типов. Например: `"123".parseInteger` возвращает целое `123`. `"12.34e-10".parseReal` возвращает вещественное значение `1.234`. `"false".toBoolean` возвращает булевское значение **`false`**.
`String`	`Real`	Нет	
`String`	`Boolean`	Нет	
`String`	`Char`	Да; явно	Допустимо посредством явного преобразования в случае, когда текущая длина строки равна единице. Результат – символ, значение которого равно (единственному) элементу строки-аргумента.

Примеры преобразований:

```
var n : Integer = 1234
var x : Real = n    // неявное преобразование целого значения к вещественному
```

```
n := Integer(12.78) // преобразование вещ. значения к целому (округление)
x := true           // ошибка: недопустимое неявное преобр-е Boolean->Real
x := Real(false)    // ошибка: недопустимое явное преобр-е Boolean->Real
```

```
Ну вот все тоже работает с конверторами … Один в один только без огромных
таблиц предопределенных преобразований …
```

Замечание. Конструкция преобразования типа на самом деле может считаться частным случаем более общей конструкции «конструирование значения», семантика которого определена для любого (как предопределенного, так и пользовательского) типа. Общий случай этой конструкции рассматривается в главе «Выражения». Здесь только заметим, что выражение вида *Тип*(*Выражение*) можно трактовать как способ задания («конструктор») нового значения заданного типа, начальным значением которого выступает значения *выражения*.

Вот теперь понял смысл! Интересно !!! беру свои слова назад!!! Посыпаю голову пеплом – молодой горячий ☺ Но с конверторами не надо писать Тип …. А просто сделать присвивание или передать подпролграмме … Т.е. синтаксически проще …

1.2 Составные типы

Как уже говорилось, составные типы можно рассматривать как способы конструирования новых типов из имеющихся. В языке определены четыре способа конструирования составных типов:

Мне не нравиться …. Но … Зачем выделять массивы, ассоциативные массивы .. Списки …

- Массивы
- Ассоциативные массивы
- Списки
- Функциональные типы

1.2.1 Массивы

Массив представляет собой одно- или многомерный набор значений одного типа, с возможностью доступа к произвольному элементу массива за фиксированное время по целочисленному индексу или (в случае многомерного массива) по набору целочисленных индексов.