А. И. Синяков*

АНАЛИЗ МОДУЛЬНОГО ПОДХОДА И ЕГО ПРИМЕНЕНИЕ В РАЗЛИЧНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

ВВЕДЕНИЕ

На сегодняшний день существует несколько подходов в программировании: модульный, объектно-ориентированный, структурный. Кратко рассмотрим все эти подходы и более подробно остановимся на модульном, а во второй части рассмотрим, как реализована поддержка модулей в некоторых языках программирования.

1. МЕТОДЫ ПРОГРАММИРОВАНИЯ

1.1. Модульный подход

Модульное программирование — это искусство разбиения задачи на некоторое число подзадач, реализуемых в виде отдельных модулей, а также умение широко использовать стандартные модули путем их параметрической настройки.

Модульность является одним из основных принципов построения программных систем. В общем случае программный модуль — это отдельная функционально законченная программная единица, некоторым образом идентифицируемая и объединяемая с другими. Понятие модульности и модульного программирования появилось давно, и до сих пор основные принципы и понятия модульного подхода не устарели и лежат в основе практически всех других подходов.

Вначале основным принципом выделения модуля было разбиение алгоритма или программы на функционально замкнутые фрагменты, используемые, как правило, неоднократно. Развитие понятия модульности сказа-

* sinal@ngs.ru

Работа выполнена при финансовой поддержке Министерства образования РФ (научная программа «Университеты России», грант № УР.04.01.201).

лось в том, что модуль стал средством декомпозиции не только структур управления, но и структур данных. Такому представлению о модуле способствовало развитие понятия типа данных. Модуль понимается не только как единица компиляции и хранения, но и как единица проектирования и раздельной разработки программной системы большим коллективом разработчиков. Таким образом, модуль понимается как средство определения логически связанной совокупности объектов, средство их выделения и изоляции.

Современный подход рассматривает модульное программирование как метод декомпозиции, в основе которого лежат структуры данных. Модуль включает в себя описание структуры данных и базовые действия над ней. Развитию такого представления о модуле в значительной мере способствовало появление понятия типа данных.

Основные аспекты модульности в языках программирования, поддерживающих модульный подход, будут рассмотрены немного позже. Композиции модулей строятся на основе информации об интерфейсе модуля. При этом под интерфейсом понимается совокупность описаний и соглашений о средствах и способах передачи объектов модулей и их подобъектов. Средства и соглашения относительно задания интерфейса модуля различны, широко используется область видимости объектов модулей путем задания списков импорта и экспорта.

Сведения об интерфейсе модуля, а также характеристики его объектов, необходимые при использовании этих объектов другими модулями, и составляют спецификацию модуля в отличие от его реализации, в которой описывается представление и алгоритмы обработки, связанные с теми или иными объектами модуля.

Рассмотрим несколько подробнее программу модульного подхода, круг решаемых им проблем.

1. Изоляция определяемого понятия

Первое и наиболее очевидное назначение модуля — выделение и изоляция некоторого понятия. Поэтому модуль в том или ином языке определяется некоторой замкнутой конструкцией, имеющей уникальное имя, для которой определены некоторые правила преобразования.

2. Введение абстрактных объектов

Введенное с помощью модуля понятие можно рассматривать как объект, имеющий определенную семантику на некотором уровне абстракции, независимо от его функционального назначения: алгоритм, тип данных, механизм распределения памяти, структура данных. Природа этих объектов

несущественна, важно, что введя такое понятие, можно пользоваться им, не вдаваясь в его представление и реализацию. Модуль хорошо подходит для описания абстрактного типа данных. Под абстрактным типом данных понимается определение некоторого понятия в виде набора из одного или нескольких объектов с некоторыми свойствами и операциями. Абстрактный тип данных задается через множество допустимых операций, поэтому главной и неотъемлемой частью модуля, описывающего абстрактный тип, должны быть перечень и описание этих операций (чаще всего в виде функций). Объекты такого типа могут использоваться как аргументы допустимых типом операций и будут тем самым защищены от некорректного или несанкционированного доступа.

3. *Модуль как средство поэтапной разработки программных систем.*

Процесс разработки систем складывается из этапов, основными из которых являются этап проектирования и этап кодирования, или программирования. На первом из них производится модуляризация проектируемой системы, т.е. разбиение ее на взаимодействующие объекты, средством описания которых и являются модули. На этом этапе каждый из модулей может быть представлен только своей краткой характеристикой или спецификацией. Это дает возможность описать сразу всю систему, оставив детальное определение объектов на более поздний срок.

Коллективность разработки крупных программных продуктов усиливает требования к описанию интерфейсов модуля, к точной фиксации не только входных и выходных данных модуля, но и таких их свойств, как тип, структура, допустимые значения. Описание входных и выходных данных модуля, условий его использования, перечень действий модуля — все это принято называть спецификацией модуля. Таким образом, в процессе разработки программных систем первостепенной задачей является именно спецификация модуля.

Полное описание модуля состоит из трех разделов: спецификация, представление и реализация (как правило, раздел представления отсутствует). Разбиение на разделы обеспечивает нечувствительность одного модуля к внутренним изменения других.

В разделе *представление* описывается внутренняя спецификация модуля. Это в основном описание структур данных, используемых при представлении объектов модуля. Выделение их в отдельный раздел позволяет ограничить к ним доступ пользователя, исключает зависимость других модулей от представления объектов данного модуля.

В разделе *реализация* дается описание действий, реализуемых модулем. Например, для модуля, определяющего некоторый абстрактный тип, это описание операций, допустимых над объектами этого типа.

4. Создание проблемно-ориентированного контекста.

Проблемно-ориентированный контекст определяется множеством объектов, необходимых для задания проблемной области, в которой ставятся задачи, решаемые данной программной системой.

5. Локализация машинной зависимости программ.

Одно из применений модульной концепции — локализация машинной зависимости внутри только отдельных модулей. Этим достигается возможность быстрой подмены машинно-зависимых фрагментов программы при переносе ее на другую платформу.

1.2. Структурный подход

Структурный подход базируется на систематическом использовании абстракций для управления массой деталей и способе документирования, который помогает проектировать программу.

В структурном подходе важна форма и дисциплина. Она нацелена на повышение продуктивности программирования за счет увеличения надежности программ и облегчения их модификации. Одним из главных способов повышения надежности является улучшение структуры программы, что позволяет понимать, сопровождать и модифицировать программу без участия авторов.

Составными частями структурного подхода являются: нисходящая разработка, структурное программирование, сквозной структурный контроль.

1. Нисходящая разработка.

Традиционно проектирование программной системы делается сверху вниз. При использовании метода нисходящей разработки детальное проектирование программ, кодирование, отладку и документирование можно делать параллельно. Нисходящая разработка призвана уменьшить сложность программы.

2. Элементы структурного программирования.

Одним из основополагающих принципов структурного программирования является то, что все создаваемые программы и их фрагменты должны быть структурированными, иначе говоря, подчиняться таким правилам композиции конструкций языка, чтобы процесс исполнения этих конструк-

ций был бы легко видим и понимаем по их текстуальному представлению: структура текста должна соответствовать структуре исполнения. Естественное требование для этого — это требование того, чтобы композиция осуществлялась такими конструкциями, исполнение которых имеет одну входную точку в тексте (с которой начинается исполнение конструкции), и одну выходную точку (исполнением которой завершается исполнение конструкции). Большинство управляющих конструкций языков построено именно таким образом. Структурированная программа не может содержать операторы перехода в произвольную точку текста: в таком случае структура текста уже не будет соответствовать структуре возможного исполнения.

3. Сквозной структурный контроль.

Сквозной структурный контроль является неотъемлемой частью структурного подхода и регламентирует организационную и контролирующую деятельность руководителей проекта. Термин «сквозной» указывает на способ проверки — все контролируемые элементы выполняются шаг за шагом.

Термин «структурный» подчеркивает, что контроль является составной частью рабочего цикла, он заранее предопределен и продуман.

Контроль ориентирован на то, чтобы обнаружить ошибки в принятых решениях и создать атмосферу, при которой сам проектировщик и другие члены проекта стремились бы найти ошибки как можно раньше.

Сквозной контроль осуществляется на всех этапах разработки, начиная с этапа определения требований.

1.3. Объектно-ориентированный подход

В основе объектно-ориентированного подхода лежит рассмотрение объекта в качестве главного понятия при решении всего круга проблем, связанных с разработкой сложных систем. Объектно-ориентированный подход (ООП) является результатом эволюционного процесса развития методологий, в частности, модульного и структурного подходов. В отличие от структурного подхода ООП подразумевает другой взгляд на процесс декомпозиции разрабатываемой системы. Как и другие парадигмы, ООП затрагивает процессы проектирования и программирования, но в отличие от других подходов он распространяется и на такие виды деятельности и типы обработки информации, как проектирование пользовательского интерфейса и архитектуры компьютеров, базы данных и базы знаний.

Главными компонентами объектного подхода является абстрагирование, ограничение доступа, модульность и иерархия.

Абстрагирование — это выделение таких существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и таким образом четко определяют особенности данного объекта с точки зрения дальнейшего его рассмотрения и анализа.

Ограничение доступа — это процесс защиты отдельных элементов объекта (структуры объекта, реализации его методов).

Модульность — это свойство системы, определяемое возможностью декомпозиции ее на ряд тесно связанных модулей. Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при проектировании системы.

Абстрагирование, ограничение доступа и модульность являются средствами упрощения сложных систем. Иерархия является эффективным механизмом сокращения сложности. В ООП структура классов является иерархией по номенклатуре, а структура объектов — иерархией по составу. Основным видом иерархии по номенклатуре является наследование. Оно означает такое соотношение между классами, когда один класс использует структурную или функциональную часть одного или нескольких классов (простое и множественное наследование).

Любой язык объектно-ориентированного программирования характеризуется тремя основными следующими свойствами.

1. *Инкапсуляция*: определение объекта путем не только описания его структуры, но и всего множества операций с ним.
2. *Наследование*: возможность построения иерархии порожденных объектов благодаря предоставлению доступа каждого из порожденных объектов к коду и данным предка.
3. *Полиморфизм*: возможность идентифицировать одним и тем же именем множество аналогичных операций (действий), аргументами которых являются разные объекты некоторой иерархии объектов.

Объектно-ориентированное программирование по существу включает в себя все понятия модульного программирования. К ним относится и понятие межмодульного интерфейса. Каждый модуль определяется как пара — определяющий раздел (или интерфейс) и реализующий раздел.

2. СРЕДСТВА МОДУЛЬНОСТИ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Рассмотрим более подробно особенности модулей в современных языках программирования и их использование.

2.1. Язык Модула 2

Модула 2 — язык программирования общего назначения, разработанный, прежде всего, для реализации систем программирования. Основывается на языках Паскаль и Модула.

Модуль предназначен для отделения спецификации от реализации, раздельной компиляции, определения новых типов и операций над ними, реализации низкоуровневых операций.

Модуль можно представить парой: определяющий и реализующий модули. Они имеют одно и то же имя. Определяющий модуль задает спецификацию доступных извне объектов модуля: констант, типов, переменных, процедур. Перечень этих объектов, кроме того, задается в списке экспорта данного модуля. Объекты, используемые в данном модуле и внешние по отношению к нему, перечисляются в списке импорта.

Синтаксис определяющего модуля имеет следующий вид:

```
definition module <имя модуля>;
        <список импорта>;
        <список экспорта>;
        <спецификация объектов>;
end <имя модуля>
```

Реализующий модуль содержит описание локальных объектов модуля и полное описание процедур и функций, специфицированных в соответствующем определяющем модуле. Все объекты, описанные в определяющем модуле (в том числе импортируемые), доступны в его реализующем модуле. Реализующий модуль может иметь тело-блок, состоящий из операторов. Если модули вложены в процедуру, а не являются единицами компиляции, то их тела исполняются в порядке следования модулей и служат для инициализации локальных переменных этой процедуры.

Синтаксис реализующего модуля таков:

```
implementation module < имя модуля >;
          <описание процедур и функций> <тело модуля>;
end < имя модуля >;
```

Определяющий модуль является интерфейсным по отношению к реализующему и использующим его модулям.

Область действия модуля определяется уровнем его описания. Если модуль задан независимо, т.е. как единица компиляции, его можно использовать в любом месте программы. Если же модуль описан внутри процедуры, он локален в теле этой процедуры. Внешние объекты модуля видимы в той же области, что и сам модуль. Внешними (доступными извне) становятся объекты, специфицированные в модуле и перечисленные явно в списке его экспорта. Модуль может потребовать уточненного (qualified) использования экспортируемых объектов заданием списка экспорта в виде

export qualified <список имен>

При использовании имена этих объектов должны предваряться именами данного модуля. Уточненный экспорт необходим при разработке модулей, которые придется использовать совместно с другими модулями, заранее не известными. Он позволяет избежать конфликтов имен объектов, экспортированных из разных модулей. В модуле возможен лишь один список экспорта, поэтому все экспортируемые объекты могут быть либо уточнены, либо нет.

Помимо описанных в модуле объектов в нем доступны также объекты, экспортируемые другими модулями, либо видимые на уровне описания модуля, когда модуль описан в некоторой процедуре. Такие объекты перечисляются в списке импорта. Объекты, импортируемые из модуля *M* и уточненные в нем, должны использоваться с префиксом *M*. Префикс можно упомянуть лишь в списке импорта, указав этим связь импортируемых объектов с именем их модуля, например,

from M *import* S_1, S_2

Рассмотрим пример модуля для работы с памятью. Пусть в программе описаны

```
type T = …;
var УК : pointer to T;
```

Операторы *new*(УК) и *dispose*(УК), служащие для создания объекта и отказа от него, будут переведены транслятором в процедуры

```
ALLOCATE(УК, TSIZE(T))
DEALLOCATE(УК, TSIZE(T)),
```

где первый параметр — указатель на размещаемый или убираемый объект, а второй — его длина.

Вместо стандартного модуля ПАМЯТЬ, где определены эти процедуры, можно описать свой модуль ПАМЯТЬ, определяющая часть которого имеет вид:

```
definition module ПАМЯТЬ;
   from SYSTEM import ADDRESS;
   export qualified ALLOCATE, DEALLOCATE, РЕЖИМ;
   procedure ALLOCATE(var УК: ADDRESS; ДЛИНА: CARDINAL);
   procedure DEALLOCATE(var УК: ADDRESS; ДЛИНА: CARDINAL);
   procedure РЕЖИМ (РЖ: CARDINAL);
      (* РЖ=1: когда не хватает свободного места,
                  ALLOCATE обрывает работу (по умолчанию);
               2: когда не хватает свободного места, ALLOCATE выделяет nil *)
                     . . .
end ПАМЯТЬ;
```

Аналогично можно определить системные или пользовательские модули управления процессами, работы с файлами и т.п.

2.2. Язык Ада

Язык Ада — универсальный язык программирования высокого уровня. Прежде всего, Ада используется для построения больших систем, к которым предъявляются достаточно высокие требования по надежности.

Пакет (модуль) — это средство, которое позволяет сгруппировать логически связанные вычислительные ресурсы и выделить их в единый самостоятельный программный модуль, реализация которого защищена от пользователя. Под вычислительными ресурсами в этом случае подразумеваются данные (типы данных, переменные, константы и т.д.) и подпрограммы, которые манипулируют этими данными.

Пакет в общем случае задается парой: спецификация и тело. Обе части имеют один и тот же идентификатор.

Спецификация определяет интерфейс с вычислительными ресурсами (сервисам) пакета, доступными для использования во внешней, по отношению к пакету, среде. Другими словами — спецификация показывает, что доступно при использовании этого пакета.

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

```
<спецификация модуля> ::=
    package <имя модуля> is
        <видимая описательная часть>
        <приватная описательная часть>
    end <имя модуля>

<тело модуля> ::=
    package body <имя модуля> is
        <описательная часть>
        begin <последовательность операторов>
        exception <реакция на исключительные ситуации>
    end <имя модуля>
```

Тело пакета может отсутствовать, если он предназначен только для описания данных или новых типов.

Если в видимой части пакета специфицированы функции или другой пакет, то тела их должны входить в описательную часть тела этого пакета. Здесь же могут быть описаны локальные объекты пакета, необходимые для реализации внешних объектов.

В конце пакета или его компонента может задаваться реакция на исключительные ситуации в виде:

```
exeption
    when <имена ситуаций> <операторы>
    when others <операторы>
```

Операторы обработки реакции имеют доступ ко всем данным компонента, в частности, к параметрам функции, и могут выполнять от имени этой функции оператор возврата.

Область видимости объектов пакета за его пределами экранируется охватывающей конструкцией (блок, программа, другой пакет). В то же время объекты, объявленные в объемлющих программных сегментах, видимы внутри данного сегмента.

Ограничить доступ можно предложением ограничения видимости (restricted), запрещающим видимость всех объектов некоторого сегмента либо разрешающим видимость указанных объектов.

Доступ к объектам видимого пакета осуществляется через составное имя:

```
<имя пакета>.<имя объекта>
```

Такой способ еще называют *использование объектов как поименованных компонентов*.

Все внешние объекты пакета можно сделать непосредственно видимыми (указанием только их имени), если в области видимости этого пакета поместить декларацию использования:

use <имя пакета>

Пример:

```
package RATIONAL NUMBERS is
    type RATIONAL is
    record
        NUMERATOR : INTEGER;
        DENOMINATOR : INTEGER range 1..INTEGER'LAST;
    end record;
    function "=" (X,Y : RATIONAL) return BOOLEAN;
    function "+" (X,Y : RATIONAL) return RATIONAL;
    function "*" (X,Y : RATIONAL) return RATIONAL;
package body RATIONAL_NUMBERS is
    procedure SAME_DENOMINATOR (X,Y : in out RATIONAL) is;
    begin
        --Задается приведение X и Y к общему знаменателю
    end;
    function "=" (X,Y : RATIONAL) return BOOLEAN is U,V: RATIONAL;
    begin
        U := X; V := Y; SAME_DENOMINATOR (U,V);
        return U.NUMERATIOR = V.NUMERATOR;
    end "=";
    function "+" (X,Y : RATIONAL) return RATIONAL is
        . . .
    end "+";
    function "*" (X,Y : RATIONAL) return RATIONAL is
        . . .
    end "*";
end RATIONAL_NUMBERS;
```

2.3. Язык Legos

Язык Legos был разработан на основе языка Паскаль для модульных систем программирования.

Модуль служит для структурирования программы, а также для указания связей между отдельными ее компонентами. Каждый модуль компилируется отдельно в контексте, создаваемом объектами, объявленными в списке видимых модулей.

В зависимости от роли в программе выделяют три типа модулей: простой, структурный и интерфейсный.

Простой модуль описывается конструкцией

module <имя модуля> *use* <имена видимых модулей>
<список деклараций> <список команд>
end <имя модуля>

В списке с ключом use перечисляются имена модулей, используемых данным, они составляют контекст для компиляции.

Структурный модуль создается с целью защитить некоторые внутренние возможности программы от модификаций. Он состоит из программы, которую необходимо закрыть, и ассоциированного с ней интерфейса. Заголовок структурного модуля имеет вид:

module <имя модуля> = K : R,

где K — специфицирующий (или интерфейсный) модуль; R — реализующий модуль.

Модуль интерфейса, являясь специфицирующим модулем, делает доступными извне объекты, реализуемые другими модулями, и определяет правила доступа к ним. Заголовок модуля интерфейса имеет следующий вид:

module <имя модуля> *of* <список имен модулей>

Модуль реализации должен содержать в своем use-списке имя своего специфицирующего модуля.

В модуле *A*, использующем модуль *B* (module *A* use *B*), могут применяться все объекты, определенные в модуле *B*.

При желании скрыть те или иные объекты модуля *B* (в различных его использованиях), может быть описано несколько его интерфейсных модулей $B_1 \ldots B_n$, содержащих спецификации только нужных объектов. Нужный для *A* интерфейс модуля *B* указывается в его списке use (т.е. module *A* use B_i). При этом видимыми становятся все специфицированные в B_i объекты.

Модули могут объединяться в конструкции, называемые программами и являющимися средством формирования контекста из объектов нескольких модулей:

program <имя программы> *of* <список имен модулей>

Для каждой такой программы можно описать один или несколько модулей интерфейса. Каждый модуль интерфейса тогда предоставляет особую видимость программы.

В то же время один и тот же модуль интерфейса может использовать различные программы (например, если каждая из них является специфической реализацией объектов, определенных в интерфейсном модуле).

Делая сборку программы, можно одновременно сопоставлять модуль и его интерфейс для данной программы. Например,

program P *of* $J_1 : M_1, J_2 : M_2, M_3$

Пример. Описание модулей для выделения и освобождения памяти в «куче».

Модуль спецификации:

```
module allocator;
    action
    allocate = procedure (inout p: pointer, size: integer);
    (* предоставление памяти длины size, заданной указателем p *)
    deallocated = procedure (inout p: pointer; size: integer);
    (* освобождение памяти длинны size с адреса p *)
end allocator;
```

Модуль реализации:

```
module allocreal use allocator, errormodule
    const m = 100;
    n = 1000;
    var first, last: array [1..m] of integer, current: integer;
    heap: array [1..n] of unspecified;
    body allocate;
        var i, j: integer;
        begin
            . . .
```

```
    end (* allocate *)
    body deallocate;
        var i, j: integer;
        begin
            . . .
    end (* deallocate *)
    begin (* инициализация *)
        current := 1;
        first [current] := 1;
        last [current] := 1000;
end allocreal;
```

2.4. Язык Clu

Основным побудительным мотивом разработки языка Clu явилась необходимость поддержки абстракции данных. Использование абстракций данных ведет к объектно-ориентированному стилю программирования, при котором данные рассматриваются как главные структуры, и это определяет организацию структуры программы. Программа на языке Clu состоит из группы модулей трех видов: процедур, итераторов (абстракция управления), кластеров (абстракция данных).

Особенность процедур языка Clu заключается в том, что в заголовке явно специфицируются исключительные условия окончания, например:

```
square_root = proc (x : real) returns (real)
    signals (no_real-result)
```

Итератор — это модуль, поставляющий последовательность элементов по их аргументам. Итератор вызывается оператором for. Поставляемый итератором элемент присваивается переменной цикла, выполняется тело цикла, управление возвращается итератору, поставляющему следующий элемент и т. д. Цикл заканчивается с завершением итератора.

Кластер используется для спецификации и реализации новых типов данных. Он реализует абстракцию данных, которая представляет собой набор объектов и элементарных операций создания этих объектов и манипулирования ими. Кластеры и итераторы могут содержать локальные собственные переменные, сохраняющие значения от вызова к вызову, но не доступные извне.

Все три вида модулей языка Clu допускают параметры. Параметризация дает возможность определять целый класс абстракций средствами одного

модуля. Параметр задается именем и типом. Помимо стандартных типов вводится тип type, означающий, что формальному параметру может соответствовать фактический параметр любого типа и модуль обеспечивает соответствующие операции с параметром произвольного типа.

Определение модуля имеет вид

имя = $\left\{\begin{matrix} proc \\ iter \\ cluster \end{matrix}\right\}$ [<параметры>] *is* <список допустимых операций>
[*where* <спецификация операций>]
<раздел представления>
<раздел описания подпрограмм>
end <имя>

Факультативная спецификация where для модуля, имеющего параметр типа type, описывает требования к соответствующим этому параметру фактическим параметрам, выраженные в терминах ограничений на операции.

Поскольку кластер описывает абстрактный тип, в его теле содержится раздел «представление» (rep), в котором в терминах некоторого конкретного типа дается описание конкретного представления этого абстрактного типа. Оно должно быть единственным для данного кластера, но может быть параметризованным.

Операции в кластере описываются в терминах конкретного представления, недоступного пользователям. Каждая операция, определяемая кластером, реализуется с помощью подпрограммы (процедуры или итератора).

Модули языка Clu никогда не вкладываются друг в друга. Программа является одноуровневой структурой, в которой каждый модуль может быть использован в любом модуле программы. Проверка типа и межмодульный интерфейс выполняются на основе информации, имеющейся в заголовке модуля, а в случае кластеров — еще и в заголовках процедур и итераторов, реализующих операции.

Каждый модуль представляет собой отдельную текстуальную единицу и компилируется независимо от других модулей.

Основными элементами семантики языка Clu являются объекты и переменные. Объектами называются единицы данных, создаваемые и обрабатываемые программами. Каждый объект имеет тип, определяющий набор элементарных операций по созданию и обработке объектов данного типа. Объект может быть создан и обработан только через эти операции.

Память под объекты выделяется динамически. Теоретически объекты существуют на протяжении всего времени исполнения программы.

В отличие от многих других языков программирования переменные в Clu не имеют значений, а используются только для ссылок на конкретные объекты во время выполнения (в частности, две переменные могут ссылаться на один и тот же объект).

Пример. Реализация типа данных «комплексное число», представимое в кластере координатами x и y.

```
complex = cluster is create, add, get_x, get_y, equal
    rep = struct [x, y: real]
    add = proc (a, b: cvt) returns (cvt)
        signals (overflow, underflow)
        return (rep $ {x: a.x+b.x, y: a.y+b.y})
        resignal overflow, underflow
    end add
    get_x = proc (c: cvt) returns (real)
        signals (overflow, underflow)
        return (c.x)
    end get_x
    get_y = proc (c: cvt) returns (real)
        signals (overflow, underflow)
        return (c.y)
    end get_y
    . . .
end complex
```

2.5. Язык Симула 67

Симула 67 — универсальный язык программирования, являющийся расширением языка Алгол 60. При его создании стремились устранить противоречия между универсальностью и специфическими требованиями к языкам для различных проблемных областей.

Понятию модуль в языке Симула 67 эквивалентно понятие класс. Центральным является понятие объект. Объект — это экземпляр блока, имеющий собственные локальные данные и действия, описанные в декларации класса. Декларация класса, не содержащая никаких действий, определяет структуру данных. Объекты создаются динамически в процессе выполнения программы.

Имя класса может быть префиксом в декларации другого класса, тогда последний становится подклассом первого. Префиксация означает, что все понятия и префиксы, определенные в префиксном классе, становятся доступными для объектов класса, имеющего этот префикс. Каждый из классов может иметь последовательность префиксов произвольной глубины. Возможности языка можно расширять, описывая так называемые системные классы. Для некоторой прикладной области можно создать системный класс, описывающий присущие ей понятия, методы и свойства.

Декларация класса имеет вид:

```
<префиксы>  class <имя класса>
              <совокупность формальных параметров>
              <спецификация параметров>
              <спецификация виртуальных величин>
              <тело класса>
            end <имя класса>
```

Декларация класса с префиксом C и идентификатором класса D (C *class* D) определяет некоторый подкласс D класса C. В общем случае, если C_1, C_2,…, C_n — классы, причем C_1 не имеет префикса, а C_k имеет префикс C_{k-1}(k = 2, 3,…), то индекс «k» называется префиксальным уровнем класса C_k; C_i является подклассом класса C_j при i > j, а префиксальный уровень подкласса C_i считается выше префиксального уровня класса C_j.

Объекты модуля называются атрибутами класса.

Рассмотрим в качестве примера модуль, описывающий составное понятие «интегрирование по Гауссу»:

```
class Гаусс(n); integer n;
   begin array W, X [1 : n];
      real procedure интеграл (F, a, b);
      real procedure F : real a, b;
      .  .  .
   end Гаусс;
```

Если переменные G5, G10 описать в виде ссылки на класс Гаусс

ref (Гаусс) G5, G10;

то после выполнения операций

G5 : — *new* Гаусс (5);
G10 : — *new* Гаусс (10)

их значениями будут объекты класса Гаусс. Тогда G5.интеграл (<параметры>) и G10. интеграл (<параметры>) называются дистанционными идентификаторами, имеющими атрибут класса.

Понятие класса равносильно понятию абстрактных типов данных. Объект конкретного типа создается операцией генерации объекта, имеющей вид

new <имя класса> (<список фактических параметров>)

Ниже представлен пример описания класса, выражающего понятие «расстановка» (функция расстановки). Процедура «оценка» объявлена виртуальной и может быть подменена другой процедурой.

```
class расстановка (n); integer n;
   virtual : integer procedure оценка;
   begin integer procedure оценка (T);
      value T; text T;
      . . .
   end оценка;
. . .
расстановка class АЛГОЛ оценка;
   begin integer procedure оценка (T);
      value T; text T;
   end (АЛГОЛ оценка);
```

Механизм виртуальных величин позволяет реализовать механизм инкапсуляции и подмены реализации объектов.

2.6. Язык Оберон 2

Оберон 2 — язык программирования общего назначения, продолжающий традиции языков Паскаль и Modula 2. Его основные черты — блочная структура, модульность, раздельная компиляция, статическая типизация со строгим контролем соответствия типов (в том числе межмодульным), а также расширение типов и связанные с типами процедуры.

Модуль — совокупность объявлений констант, типов, переменных и процедур вместе с последовательностью операторов, предназначенных для присваивания начальных значений переменным. Модуль представляет собой текст, который является единицей компиляции.

```
module = MODULE indent ";" [ImportList] DeclarationSequence
        [BEGIN StatementSequence] END ident "."
ImportList = IMPORT import {"," import} ";"
Import = ident [":=" ident]
```

Список импорта определяет имена импортируемых модулей. Если модуль *A* импортируется модулем *M* и *A* экспортирует идентификатор *x*, то *x* упоминается внутри *M* как *A.x*. Если *A* импортируется как *B:=A*, объект *x* должен вызываться как *B.x*. Это позволяет использовать короткие имена-псевдонимы в уточненных идентификаторах. Модуль не должен импортировать себя. Идентификаторы, которые экспортируются (т.е. должны быть видимы в модулях-импортерах) нужно отметить экспортной меткой в их объявлении.

Последовательность операторов после символа BEGIN выполняется, когда модуль добавляется к системе (загружается). Это происходит после загрузки импортируемых модулей. Отсюда следует, что циклический импорт модулей запрещен. Отдельные (не имеющие параметров и экспортированные) процедуры могут быть активированы из системы.

Оберон 2 поддерживает динамическую загрузку модулей. Загруженный модуль может вызывать команду незагруженного модуля, задавая ее имя как строку. Специфицированный модуль при этом динамически загружается и выполняется заданная команда. Динамическая загрузка позволяет пользователю запустить программу как небольшой набор базисных модулей и расширять ее, добавляя последующие модули во время выполнения по мере необходимости.

Модуль M_0 может вызвать динамическую загрузку модуля M_1 без того, чтобы импортировать его. M_1 может, конечно, импортировать и использовать M_0, но M_0 не должен знать о существовании M_1. M_1 может быть модулем, который спроектирован и реализован намного позже M_0.

Пример:

```
MODULE Trees; (* экспорт: Tree, Node, Insert, Search, Write, Init *)
   IMPORT Texts, Oberon; (* экспорт только для чтения: Node.name *)
   TYPE
      Tree* = POINTER TO Node;
      Node* = RECORD
         name-: POINTER TO ARRAY OF CHAR;
         left, right: Tree
```

```
        END;
    VAR w: Texts.Writer;
    PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
        VAR p, father: Tree;
    BEGIN p := t;
        REPEAT father := p;
            IF name = p.name^ THEN RETURN END;
            IF name < p.name^ THEN p := p.left ELSE p := p.right END
        UNTIL p = NIL;
        NEW(p);
        p.left := NIL;
        p.right := NIL;
        NEW(p.name, LEN(name)+1);
        COPY(name, p.name^);
        IF name < father.name^ THEN father.left := p ELSE father.right := p END
    END Insert;
    PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree;
        VAR p: Tree;
    BEGIN p := t;
        WHILE (p # NIL) & (name # p.name^) DO
            IF name < p.name^ THEN p := p.left ELSE p := p.right END
        END;
        RETURN p
    END Search;
    PROCEDURE (t: Tree) Write*;
    BEGIN
        IF t.left # NIL THEN t.left.Write END;
        Texts.WriteString(w, t.name^);
        Texts.WriteLn(w);
        Texts.Append(Oberon.Log, w.buf);
        IF t.right # NIL THEN t.right.Write END
    END Write;
    PROCEDURE Init* (t: Tree);
    BEGIN NEW(t.name, 1);
        t.name[0] := 0X;
        t.left := NIL;
        t.right := NIL
    END Init;
BEGIN Texts.OpenWriter(w)
END Trees.
```

2.7. Язык Perl

Perl — интерпретируемый язык, приспособленный для обработки произвольных текстовых файлов, извлечения из них необходимой информации и выдачи сообщений. Perl также удобен для написания различных системных программ. Этот язык прост в использовании, эффективен, но про него трудно сказать, что он элегантен и компактен. Perl сочетает в себе лучшие черты C, shell, sed и awk, поэтому для тех, кто знаком с ними, изучение Perl не представляет особого труда. Синтаксис выражений Perl близок к синтаксису C. Хотя Perl приспособлен для сканирования текстовых файлов, он может обрабатывать также двоичные данные и создавать .dbm файлы, подобные ассоциативным массивам. Perl позволяет использовать регулярные выражения, создавать объекты, вставлять в программу на C или C++ куски кода на Perl, а также позволяет осуществлять доступ к базам данных, в том числе Oracle.

Хотя в языке Perl и существует понятие модуль (package), синтаксис соответствующей конструкции и, вообще, само понятие определить четко (в виде РБНФ, например) в общем случае довольно сложно. Дело в том, что, в отличие, скажем, от языков с традиционно развитыми средствами модульного программирования, вроде Modula и Oberon, в которых модуль — это статическая конструкция, где каждый элемент имеет свое, четко определенное назначение и место, в Perl модуль — это, скорее, даже не законченная синтаксическая конструкция, а некий конгломерат объявлений, функций, операторов, выражений, директив импорта и тэгов.

Не следует думать, что модуль — это обязательная конструкция для оформления программ на Perl. Если о языке Oberon можно сказать, что любая программная система, написанная на нем, есть совокупность модулей, то в отношении языка Perl такого сказать нельзя: программная система на нем в общем случае состоит из модулей, классов (рассматриваемых как некая разновидность модуля) и скриптов (файлов, содержащих тело модуля без заголовка).

Синтаксис оформления модуля на Perl выглядит следующим образом:

```
package SomeModule;
  # Все, что следует после заголовка модуля, считается телом модуля,
  # которое оканчивается заголовком другого модуля или концом файла
  # Можно, хотя это необязательно, в конце модуля поставить тэг __END__
  # или тэг __DATA__, после которых и до конца файла любой текст будет
  # проигнорирован компилятором.
```

В модулях Perl синтаксисом не предусмотрены, как это сделано в Modula и Oberon, специальные разделы для описания списков импорта и экспорта, типов, констант, переменных, функций, инициализации. Блоки инициализации и завершения, директивы импорта, операторы и выражения, описания функций могут следовать в любом порядке, практически без ограничений. Переменные могут использоваться до их объявления и инициализации, а функции — до их описания, и такие ситуации не контролируются компилятором — все они обрабатываются на этапе выполнения.

Perl допускает множественное определение функции с одним и тем же именем в области видимости одного модуля. При вызове функции по этому имени управление будет передано той, что была описана последней, а сообщение об ошибке не появится ни на этапе компиляции, ни на этапе выполнения. Данная "особенность", совместно с неспособностью Perl проконтролировать наличие вызываемой функции на этапе компиляции, способна несколько осложнить отладку.

Данную ситуацию не следует путать с механизмом перегрузки функций в стиле C++: существование данного механизма, основанного на различиях в описании принимаемых перегруженными функциями параметров, исключено в Perl, поскольку средства описания функций Perl слишком примитивны.

В Perl существует несколько имен, которые имеют предопределенное назначение и используются интерпретатором во время выполнения для тех или иных целей. Здесь рассмотрим три функции, предназначенные для описания: `BEGIN`, `END` и `AUTOLOAD`. Поскольку ключевое слово `sub` при этом указывать необязательно, то они часто называются не функциями, а блоками.

Блоки `BEGIN` и `END` — это соответственно блоки инициализации («конструкторы») и завершения («деструкторы») модуля. «Конструкторы» выполняются сразу же, как только они обработаны компилятором, даже не дожидаясь разбора остальной части модуля. Это, как правило, происходит на этапе загрузки модуля в память при запуске программы (если модуль импортирован директивой `use`) или при динамической компиляции и загрузке в память на этапе выполнения программы (если модуль импортирован директивой `require`). «Деструкторы» выполняются при завершении работы интерпретатора. В одном модуле может быть описано множество блоков `BEGIN` и `END`, но, в отличие от пользовательских функций, все они в нужное время будут вызваны интерпретатором, причем блоки `BEGIN` вызываются в порядке их описания, а блоки `END` — в порядке,

обратном порядку их описания в модуле. Такой порядок вызова принят для того, чтобы соответствующие «конструкторы» и «деструкторы» можно было группировать парами и обеспечить при этом вызов «деструкторов» в обратном «конструкторам» порядке:

```
begin {
   # вызван первым
}
end {
   # вызван последним
}
begin {
   # вызван вторым
}
end {
   # вызван предпоследним
}
. . .
```

Блоки AUTOLOAD были введены в язык как средство борьбы с неэффективностью запуска программы, возникающей при большом количестве модулей в ней. Дело в том, что Perl не позволяет откомпилировать модули предварительно в псевдокод, а затем использовать уже откомпилированные версии для запуска и выполнения программы. Вместо этого компиляция модулей происходит при каждом запуске программы (если модуль импортирован директивой `use`) или в процессе выполнения (если модуль импортирован директивой `require`). Если определение вызываемой функции в модуле не найдено, то управление будет передано блоку AUTOLOAD (если он есть, в противном случае произойдет ошибка этапа выполнения). В качестве параметров ему будут переданы те, что были указаны при вызове функции, а встроенная переменная $AUTOLOAD будет содержать квалифицированное имя вызываемой функции.

Блок AUTOLOAD может обрабатывать такие вызовы по своему усмотрению, в частности, загружая на этапе выполнения определения необходимых функций из внешних файлов. На механизме автозагрузки основана работа стандартных модулей `AutoLoader` и `SelfLoader`, которые позволяют, разделив описания функций одного модуля по разным файлам, загружать их по требованию.

Данный механизм, по сути, является избыточным, поскольку в Perl уже есть средство динамической компиляции и загрузки модуля — это директива `require`. Автозагрузка просто позволяет распространить этот подход на уровне отдельных функций модуля, что может быть оправданным только в том случае, когда модуль содержит большое количество больших функций.

Синтаксисом модуля языка Perl не предусмотрена возможность описания списка экспортируемых объектов. Модуль в Perl может экспортировать лишь переменные и функции, поскольку возможность не то что экспорта, но даже описания констант и типов отсутствует. По умолчанию все переменные и все функции являются экспортируемыми, т. е. доступными из внешних модулей, импортирующих данный модуль.

Для объявления частных переменных модуля можно использовать встроенную функцию `my`, поскольку модуль неявно определяет блок, в котором содержится его тело и которому соответствует своя область видимости. Переменные, объявленные с помощью `my`, доступны только внутри данного модуля. А вот для описания частных функций модуля соответствующих средств нет — приходится описывать частную переменную-ссылку на анонимную функцию и вызывать ее по ссылке:

```
my $PrivateFunction = sub {
   # тело функции
};
```

Для импорта модулей предназначены две директивы: `use` и `require`. Первая используется для статического импорта (на этапе компиляции), а вторая — для динамического (на этапе выполнения). Директива `use`, кроме имени импортируемого модуля, позволяет указать список импортируемых объектов, например:

```
use SomeModule ("SomeVariable", "SomeFunction");
```

Если список указан и он не пуст либо если список не указан вовсе, то модуль будет откомпилирован, загружен, и будет вызвана функция `import`, которая может быть определена в импортируемом модуле и обязана самостоятельно реализовать импорт запрашиваемых объектов, имена которых ей будут переданы в качестве параметров. Собственно Perl эту возможность не поддерживает, а если функция `import` в модуле не оп-

ределена, то ошибки не возникнет ни на этапе компиляции, ни на этапе выполнения. При этом функция `import` фактически может реализовать не импорт, а любое другое действие. Если указан пустой список, то ни один объект не будет импортирован, а функция `import` вызвана не будет.

Чтобы облегчить жизнь разработчику модулей, в стандартную библиотеку Perl включен модуль `Exporter`, который реализует функцию `import`. Ее и следует использовать в подавляющем большинстве случаев. Здесь мы не будем подробно останавливаться на этом модуле. Заметим лишь, что для того чтобы воспользоваться всеми преимуществами модуля `Exporter`, ваш модуль должен наследовать модуль `Exporter`, рассматриваемый в данном случае как класс.

Надо отметить, что под словом «импорт» в Perl понимается нечто отличное от того, что принято понимать под этим словом в других языках. «Импорт» означает, что при обращении к импортированным объектам можно не использовать квалифицированных идентификаторов, а работать с ними так, будто они описаны в данном модуле. При этом сохраняется возможность обращения к прочим доступным объектам (не my-переменным и всем функциям) с помощью квалифицированных идентификаторов, например:

```
SomeModule::SomeFunction.
```

Таким образом, импорт не позволяет на самом деле управлять видимостью объектов при разработке модуля.

Наконец, отметим, что все имена, определенные в модуле, хранятся в хеше с именем, совпадающим с именем модуля, с «присоединенным» к нему символом «::». Причем этот хеш и его элементы доступны как для чтения, так и для записи. Возможностью «пополнения» таблицы символов модуля активно пользуются такие стандартные модули Perl, как `Exporter` и `overload`.

2.8. Язык Haskell

Haskell является одрим из наиболее мощных функциональных языков. Наиболее важными возможностями Haskell являются следующие.

- Haskell — ленивый (non-strict) язык.

- Haskell — чисто функциональный (т.е. не содержащий конструкций, неявно зависящих от состояния среды или изменяющих ее, например, он не содержит оператор присваивания).
- Haskell содержит простой и логичный механизм перегрузки функций (известный как «классы типов»). Возможные применения этого механизма выходят далеко за рамки решения непосредственной задачи. К примеру, классы типов обеспечивают возможности близкие (в целом, превосходящие) возможностям шаблонов C++, сохраняя при этом возможность раздельной трансляции.
- Императивные возможности реализуются в Haskell при помощи так называемых монад. Понимание этой конструкции является одной из наибольших проблем при изучении Haskell, но она стоит того, чтобы с ней разобраться.

В Haskell модули несут двоякое назначение — с одной стороны, модули необходимы для контроля над пространством имен (как, собственно, и во всех других языках программирования), с другой стороны, при помощи модулей можно создавать абстрактные типы данных.

Определение модуля в Haskell достаточно просто. Именем модуля может быть набор любых символов; имя начинается только с заглавной буквы. Дополнительно имя модуля никак не связано с файловой системой (как, например, в Pascal и Java), т.е. имя файла, содержащего модуль, может быть не таким же, как и название модуля. На самом деле, в одном файле может быть несколько модулей, так как модуль — это всего лишь декларация самого высокого уровня. Как известно, на верхнем уровне модуля в Haskell может быть множество деклараций (описаний и определений) — типы, классы, данные, функции. Однако один вид деклараций должен стоять в модуле на первом месте (если этот вид деклараций вообще используется). Речь идет о включении в модуль других модулей — для этого используется служебное слово *import*. Остальные определения могут появляться в любой последовательности.

Определение модуля должно начинаться со служебного слова *module*. Например, ниже приведено определение модуля Tree:

```
module Tree (Tree (Leaf, Branch), fringe) where
  data Tree a = Leaf a
  Branch (Tree a) (Tree a)
  fringe :: Tree a -> [a]
  fringe (Leaf x) = [x]
  fringe (Branch left right) = fringe left ++ fringe right
```

В этом модуле описан один тип; вполне нормально, что имя типа (Tree) совпадает с названием модуля, в данном случае они находятся в различных пространствах имен) и одна функция (fringe). В данном случае модуль Tree явно экспортирует тип Tree (вместе со своими подтипами Leaf и Branch) и функцию fringe — для этого имена типа и функции указаны в скобках после имени модуля. Если наименование какого-либо объекта не указывать в скобках, то он не будет экспортироваться, т.е. этот объект не будет виден извне текущего модуля.

Использование модуля в другом модуле выглядит следующим образом:

```
module Main where
import Tree (Tree(Leaf, Branch), fringe)
main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

В приведенном примере видно, что модуль Main импортирует модуль Tree, причем в декларации import явно описано, какие именно объекты импортируются из модуля Tree. Если это описание опустить, то импортироваться будут все объекты, которые модуль экспортирует, т.е. в данном случае можно было просто написать: import Tree.

Бывает так, что один модуль импортирует несколько других (надо заметить, что это обычная ситуация), но при этом в импортируемых модулях существуют объекты с одним и тем же именем. Естественно, что в этом случае возникает конфликт имен. Чтобы этого избежать, в Haskell существует специальное служебное слово *qualified*, при помощи которого определяются те импортируемые модули, имена объектов в которых приобретают вид: <Имя Модуля>.<Имя Объекта>, т.е. для того чтобы обратиться к объекту из квалифицированного модуля, перед его именем необходимо написать имя модуля:

```
module Main where
import qualified Tree
main = print (Tree.fringe (Tree.Leaf 'a'))
```

Использование такого синтаксиса полностью лежит на совести программиста. Некоторым нравится полная определенность, которую привносят квалифицированные имена, и они используют их в ущерб размеру программ. Другим нравится использовать короткие мнемонические имена, и они используют квалификаторы (имена модулей) только по необходимости.

В Haskell только с помощью модуля можно создать так называемые абстрактные типы данных, т.е. такие, в которых скрыто представление типа, но открыты только специфические операции над созданным типом, набор которых вполне достаточен для работы с ним. Например, хотя тип Tree является достаточно простым, его все-таки лучше сделать абстрактным типом, т.е. скрыть то, что Tree состоит из Leaf и Branch. Это делается следующим образом:

```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf) where
  data Tree a = Leaf a
  Branch (Tree a) (Tree a)
  leaf = Leaf
  branch = Branch
  cell (Leaf a)    = a
  left (Branch l r) = l
  right (Branch l r) = r
  isLeaf (Leaf _) = True
  isLeaf _ = False
```

Видно, что внешний пользователь (программист) может получить доступ к внутреннему содержанию типа Tree только при помощи использования определенных функций. Впоследствии, когда создатель этого модуля захочет изменить представление типа (например, оптимизировать его), ему необходимо будет изменить и функции, которые оперируют полями типа Tree. В свою очередь, программист, который использовал в своей программе тип Tree, ничего менять не будет, так как его программа все также останется работоспособной.

В декларации импорта (import) можно выборочно спрятать некоторые из экспортируемых объектов (при помощи служебного слова *hiding*). Это

бывает полезным для явного исключения определений некоторых объектов из импортируемого модуля.

При импорте можно определить псевдоним модуля для квалификации имен экспортируемых из него объектов. Для этого используется служебное слово *as*. Это может быть полезным для укорачивания имен модулей.

Все программы неявно импортируют модуль Prelude. Если сделать явный импорт этого модуля, то в его декларации возможно скрыть некоторые объекты, чтобы впоследствии их переопределить.

Все декларации *instance* неявно экспортируются и импортируются всеми модулями.

Методы классов могут быть так же, как и подтипы данных, перечислены в скобках после имени соответствующего класса во время декларации экспорта/импорта.

2.9. Язык Sisal 3.0

Sisal 3.0 является функциональным языком программирования. Он ориентирован на поддержку научных вычислений и представляет собой дальнейшее развитие языка Val. Обладая всеми качествами, присущими функциональным языкам программирования, Sisal способствует разработке корректных детерминированных программ, которые свободны от совмещения имен, побочных эффектов и ошибок, зависящих от реального времени. Результаты детерминированы, невзирая на архитектуру, операционную систему или обстановку исполнения.

Модуль в языке Sisal представляет собой независимую единицу компиляции, состоящую из двух частей: интерфейса (interface) и реализации (implementation). Интерфейсная часть модуля предназначена для определения функций типов и редукций. Реализационный раздел содержит полное описание функций и редукций из интерфейса модуля. Кроме того, в нем могут определяться дополнительные объекты (функции, типы, константы), необходимые для реализации.

Синтаксис интерфейса выглядит следующим образом:

```
interface <имя интерфейса>
        <объявление функций>
        <определение типов>
        <объявление редукций>
```

Реализация имеет синтаксис

```
implementation <имя реализации>
        <полные определения функций и редукций из соответствующего интерфейса>
        <другие определения>
```

Интерфейс определяет связи модуля с внешним миром. Подключение модуля к программе (другому модулю) выполняется при помощи ключевого слова uses

```
uses <имя модуля>
```

Программы на языке Sisal — это совокупность модулей, связанных друг с другом при помощи объявлений экспорта и импорта. Среди совокупности функций из всех задействованных модулей должна быть одна, с которой начинается исполнение программы.

Объявление импорта производится после имени интерфейса перед всеми определениями либо после имени реализации, перед описанием всех объектов. При этом вызов какого-либо объекта из подключенного модуля осуществляется обращением к имени этого объекта. Если в программе уже присутствует такое имя, то при трансляции будет сообщено о семантической ошибке. В таком случае в языке предусмотрена возможность составного имени для вызова объектов модуля

```
<имя модуля>.<имя объекта>
```

Такой синтаксис позволяет избежать проблемы совпадения имен при использовании модулей. Использование расширенного синтаксиса разрешается даже в случае однозначности имени, но это будет переопределением.

Если модуль написан на другом языке программирования, то для его подключения необходимо выделить из dll-файла этого модуля его интерфейс. При написании подобных модулей необходимо учитывать специфику типов языка Sisal. Наглядным примером является ошибочное значение, которым дополнено множество значений каждого типа данных в языке.

Модуль может храниться как исходный файл (здесь и далее под файлом модуля подразумевается оба файла, описывающих данный модуль), как внутреннее представление (файл содержит запись откомпилированного исходного модуля) либо в виде dll-файла.

При подключении модуля разрешается использовать любую из вышеперечисленных форм. Предполагается использовать динамический способ загрузки модулей, т.е. модуль подгружается только в момент его вызова, а не при начальном запуске программы. При вызове функции из модуля может произойти одно из следующих действий: если функция берется из файла внутреннего представления, то производится проверка версии исходного модуля и объектного файла (при обнаружении несоответствия производится перекомпиляция этого модуля) и после этого создается новый объект внутреннего представления со ссылкой на этот метод. Либо, если имеется только исходный файл модуля, вначале производится его трансляция в объектный код, а затем вызов, как и в предыдущем случае. При использовании dll-файла из его интерфейса извлекаются входные и выходные параметры запрашиваемой функции, по которым и создается объект внутреннего представления.

СПИСОК ЛИТЕРАТУРЫ

1. **Бежанова М.М., Чилингарова Т.П**. Модуляризация в современных языках программирования. — Новосибирск, 1983. — 45 с. — (Препр. / АН СССР, Сиб. отд-ние ВЦ; N 453).
2. **Математическое** обеспечение ЭВМ: окружения и интерфейсы / Сост. И.В. Поттосин, М.М. Бежанова; Новосиб. гос. ун-т. — Новосибирск, 1994. — 74 с.
3. **Programming** in Ada 95 / J.Barnes — Beading etc.: Addison-Wesley; 2nd ed., 1998. — 720 p.
4. **Nikitin E.** Into the realm of Oberon: An introduction to programming and the Oberon-2 programming language. — Berlin etc.: Springer-Verlag, 1998. — 197 p.

5. **Wall L., Christiansen T., Schwartz R**. Programmig Perl. — O'Reilly, 2nd ed., 1996. — 670 p.
6. **Haskell**: The craft of functional programming / S.Thompson. — Beading etc.: Addison Wesley; 2nd ed., 1999. — 512 p.
7. **Поддержка** супервычислений и интернет-ориентированные технологии: Сб. статей / Под ред. В.Н. Касьянова; АН СССР. Сиб. отд-ние. Ин-т систем информатики. — Новосибирск, 2001. — 264 с.
8. **Стасенко А.П.** Внутреннее представление системы функционального программирования Sisal 3.0. — Новосибирск, 2004. — 84 с. — (Препр. / СО РАН. Ин-т систем информатики; № 110).