В.Н. Касьянов, А.П. Стасенко

ЯЗЫК ПРОГРАММИРОВАНИЯ SISAL 3.2[1]

ВВЕДЕНИЕ

Используя традиционные языки и методы, очень трудно разработать высококачественное, переносимое программное обеспечение для параллельных компьютеров. В частности, параллельное программное обеспечение не может быть разработано с малыми затратами на последовательных компьютерах и потом перенесено на параллельные вычислительные системы без существенного переписывания и отладки. Поэтому высококачественное параллельное программное обеспечение может разрабатываться только небольшим кругом специалистов, имеющих прямой доступ к дорогостоящему оборудованию. Однако, используя языки программирования с неявным параллелизмом, такие как функциональный язык Sisal (аббревиатура с английского выражения Streams and Iterations in a Single Assignment Language) [1], можно преодолеть этот барьер и предоставить широкому кругу специалистов, которые не имеют доступа к параллельным вычислительным системам, но могут многое сделать в своих прикладных областях исследований, возможность быстрой разработки переносимых параллельных алгоритмов на своем рабочем месте.

Функциональная семантика языков программирования с неявным параллелизмом гарантирует детерминированные результаты для параллельной и последовательной реализаций — то, что невозможно гарантировать для традиционных языков, подобных языку Фортран. Более того, неявный параллелизм языка снимает необходимость переписывания исходного кода при переносе его с одного компьютера на другой. Гарантировано, что программа с неявным параллелизмом, правильно исполняющаяся на персональном компьютере, будет также давать правильные результаты на высокоскоростном параллельном или распределенном вычислителе.

Статья содержит описание текущей версии входного языка Sisal 3.2 системы функционального программирования SFP [2], работа над которой ведется в Лаборатории конструирования и оптимизации программ Института систем информатики СО РАН имени А.П. Ершова.

[1] Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 07-07-12050).

Цель проекта — предоставить прикладному программисту на его рабочем месте удобную среду для разработки функциональных программ, предназначенных для последующего исполнения на параллельных вычислительных системах, доступных через телекоммуникационные сети. В рамках этой среды программист получает возможность, с одной стороны, создавать и отлаживать программу без учета целевой параллельной архитектуры, а с другой — производить настройку отлаженной программы на ту или другую целевую параллельную архитектуру для достижения высокой эффективности исполнения разработанной программы на супервычислителе.

Язык программирования Sisal является одним из самых известных потоковых языков промышленного уровня и позиционируется как замена языка Фортран для научных применений [3]. Язык Sisal является результатом сотрудничества Ливерморской национальной лаборатории имени Лоренца, университета штата Колорадо, Манчестерского университета и корпорации DEC. Последняя спецификация языка Sisal версии 2.0 [4] датируется 1991 г. В 1995 г. появилось пользовательское описание языка Sisal 90 [5, 6], не содержащее точных спецификаций языка.

Язык Sisal имеет следующие особенности, облегчающие переход с популярных императивных языков программирования: приближенный к языку Паскаль [7] синтаксис, развитую систему типов и явно выделенные циклические выражения. Язык Sisal имеет следующие основополагающие качества: математическая правильность функций (отсутствие побочных эффектов), прозрачность ссылок имен, задающих значения, а не ячейки памяти, и однократность присваивания.

В 2001 г. в Институте систем информатики (ИСИ) имени А. П. Ершова СО РАН была разработана концепция языка Sisal 3.0 [8], развивающая язык Sisal 90, которая предполагала поддержку расширенных межмодульных взаимодействий, мультиязыкового программирования, а также возможностей предварительной обработки (preprocessing) и аннотирования для упрощения оптимизирующих преобразований.

В языке Sisal 3.1 [9] были специфицированы средства расширенных межмодульных взаимодействий и предварительной обработки программ. Вопросы, связанные с описанием мультиязыкового программирования и аннотирования программ языка Sisal 3.0, были оставлены для внедрения в последующих версиях языка. В язык были добавлены новые средства: пользовательские типы, переопределение операций и перегрузка функций. Базовая часть языка Sisal 90 была переработана для повышения её пригодности к нисходящему синтаксическому разбору и приближения её сходства

с языком Си. Для языка Sisal 3.1 был реализован компилятор переднего плана во внутреннее представление IR1 [10].

В языке Sisal 3.2, рассматриваемом в данной статье, специфицируется часть аннотаций (прагм), а идея поддержки мультиязыкового программирования выразилась во введении инородных типов и спецификации интерфейсов инородных модулей. Также в язык были добавлены возможности задания многомерных массивов, пользовательских типов с параметрами и обобщенных процедур. Также на язык в значительной мере повлияли идеи языка Sisal 2.0, которые не были использованы в языке Sisal 90.

Статья имеет следующую структуру. В разделе 1 рассматривается общая структура программы языка Sisal 3.2. В разделах 2 и 3 более подробно рассматриваются структура интерфейса модуля и структура модуля соответственно. Раздел 4 посвящен описанию типов и их операций, а в разделе 5 находится описание выражений. В разделе 6 описывается интерфейс взаимодействия с другими языками. В разделе 7 приводится строгое описание синтаксической и лексической структуры языка, а в разделе 8 находится писание формата Fibre для внешних значений, которые получает и порождает программа на языке Sisal. В разделе 9 обзорно приводится структура единицы компиляции на языке Си++, в которую предполагается транслировать модуль на языке Sisal.

1. СТРУКТУРА ПРОГРАММЫ ЯЗЫКА SISAL 3.2[2]

Программой (program) на языке Sisal называется совокупность файлов, каждый из которых является модулем (module) или интерфейсом модуля (module interface). Интерфейс модуля соответствует одному модулю программы, каждый из которых может иметь не более одного интерфейса.

Модуль на языке Sisal содержит определения и объявления процедур, типов (type) и определения контрактов (contract). Под процедурой понимается функция (function) или операция (operation). Модуль может быть задан:

- файлом с расширением «.s», содержащим текст на языке Sisal;
- файлом с расширением «.ir1»[3], содержащим объекты внутреннего представления IR1 (internal representation one);

[2] Далее, если не указано обратного, подразумевается версия языка Sisal.

[3] Компилятор переднего плана (front-end) Sisal порождает файл с расширением «.ir1» из файла с расширением «.s».

– файлом с расширением «.ir1.cpp»[4], который содержит исходный код на языке Си++ [11], удовлетворяющий требованиям раздела 9;
– бинарным объектным файлом с расширением «.obj»[5], сгенерированным компиляторами Си[6] или Фортран.

Модуль в формате «.ir1» описывает модуль на языке Sisal без потерь и позволяет избежать повторного анализа текста на языке Sisal. Модуль в формате «.ir1.cpp» теряет потоковую структуру программы, но полностью сохраняет возможности межмодульного взаимодействия с другими модулями. Модуль в формате «.obj» может поддерживать ограниченные возможности по взаимодействию с другими модулями, описанные в разделе 6.

Интерфейс модуля содержит видимые извне объявления процедур, определяемых в модуле, которому этот интерфейс соответствует. Также интерфейс модуля может содержать видимые извне объявления и определения типов и определения контрактов. Интерфейс модуля может быть задан:
– файлом с расширением «.s», содержащим текст на языке Sisal;
– файлом с расширением «.ir1».

Программа начитает исполняться с вызова функции main языка Си. Если функция main находится в модуле, транслируемом из представления IR1, то к нему добавляется функция «main» языка Си, которая преобразует входной поток символов формата Fibre, описываемого в разд. 8, во входные параметры функции main представления IR1. Результаты функции «main» выводятся в выходной поток символов в формате Fibre.

2. СТРУКТУРА ИНТЕРФЕЙСА МОДУЛЯ

Интерфейс модуля на языке Sisal начинается с ключевого слова[7] *interface*[8], за которым следует его имя, задаваемое идентификатором[9]. Это имя совпадает с именем модуля, которому данный интерфейс сопоставлен. В программе не может быть модулей с одинаковыми именами. Имена модулей, заданных файлами «.ir1.cpp» и «.obj», определяются на уровне проекта программы.

[4] Компилятор SFP порождает файл с расширением «.ir1.cpp» из файла с расширением «.ir1».

[5] В операционной системе Linux объектный файл имеет расширение «.o».

[6] Под языком Си всюду понимается подмножество языка Си++, соответствующее языку Си.

[7] Список ключевых слов находится в разделе 7.3.

[8] В тексте документа ключевые слова выделяются курсивом.

[9] Определение идентификатора находится в разделе 7.3.

Для модулей, заданных объектным файлом «.obj», после имени его интерфейса может следовать ключевое слово *in*, за которым находится имя языка единицы компиляции (compilation unit) данного объектного файла. Указание языка накладывает специфику на правила задания интерфейса модуля, описанную в разделе 6. Если язык не указан, то считается, что данный объектный файл был скомпилирован из файла с расширением «.ir1.cpp», и интерфейс модуля задаётся так, как описано в данном разделе.

Интерфейс модуля может содержать объявления процедур, объявления и определения типов, определения контрактов, а также конструкции импорта определений и объявлений типов и определений контрактов из других интерфейсов модулей[10]. Содержимое интерфейса модуля может разделяться точкой с запятой. Все объявления и определения влияют только на последующий текст интерфейса модуля. Интерфейс модуля завершается ключевыми словами «*end interface*».

Например, далее приводится пример интерфейса модуля «math», содержащего объявления функций для вычисления факториала и числа Фибоначчи (текст после двойной косой черты является комментарием до конца строки):

```
interface math
  function fact (n:integer returns integer) // факториал числа n
  function fib (n:integer returns integer) // число Фибоначчи с номером n
end interface
```

2.1. Объявления и определения типов

Определить можно переименованный и пользовательский типы. Определение переименованного типа выглядит как «*type* имя типа[11] = базовый тип», а определение пользовательского типа выглядит как «*type* имя типа := базовый тип». Язык Sisal поддерживает определение (как пользовательского типа) рекурсивных объединений (*union*); таким образом, имя определяемого объединения может участвовать в типах его тегов. Однако хотя бы один тип тега не должен зависеть от имени определяемого объединения во избежание бесконечных типов.

[10] В интерфейс модуля **A** импортируются только типы, так как функции, операции и редукции интерфейса модуля **B** непосредственно не могут быть использованы в интерфейсе модуля **A**. Также, при импортировании интерфейса модуля **A** было бы ненаглядно с помощью необъявленного имени интерфейса модуля **B** получать доступ к его содержимому. С другой стороны, также ненаглядно рассматривать объекты модуля **B** в качестве объектов модуля **A**.

[11] Подчеркиванием обозначается имя метапонятия.

```
// запись двух вещественных полей
type complex_record = record [ real, imag: real ]
// тип комплексного числа
type complex := record [ real, imag: real ]
// список целых чисел
type listi := union [ empty; item: record [value: integer; next: listi] ]
```

Объявить можно пользовательский тип с параметрами. Объявление пользовательского типа с параметрами[12] выглядит как «*type* имя типа [список имён параметров[13]] := базовый тип», где в базовом типе должны быть использованы все указанные имена параметров[14]. Определение пользовательского типа с параметрами выглядит как «имя типа [список типов параметров]».

```
// список произвольных элементов
type list[T] := union [ empty; item: record [value: T; next: list] ]
```

Для встроенных, составных и переименованных типов используется структурная эквивалентность базовых типов[15], для пользовательских типов используется эквивалентность по имени типа и имени модуля, в котором он был определён или объявлен, а для определённых пользовательских типов с параметрами дополнительно требуется эквивалентность типов соответствующих параметров. Пользовательские типы не эквивалентны встроенным, составным и переименованным типам. Пользовательские типы с параметрами не эквивалентны пользовательским типам без параметров.

```
// тип complex_record2 эквивалентен типу complex_record
type complex_record2 = record [ real2: real; imag2: real ]
// тип complex_record3 не эквивалентен типу complex_record
type complex_record3 = record [ real3: integer; imag3: integer ]
// типы someint из модулей A и B не эквивалентны
interface A type someint := integer end interface
interface B type someint := integer end interface
// тип listi не эквивалентен типу list[integer]
```

[12] Пользовательский тип с параметрами служит заменой понятия множества типов из предыдущих версий языка Sisal. Множество типов при определении фиксирует входящие в него типы, что не позволяет использовать обобщенные алгоритмы (функции с формальными параметрами, являющимися множествами типов) для других типов. Также пользовательский тип с параметров позволяет реализовывать пользовательские составные типы, что невозможно с помощью множеств типов. Функциональность рекурсивных множеств типов языка Sisal 90 восполняется рекурсивными объединениями.

[13] Имена параметров должны быть различны. По умолчанию, элементы списка разделяются запятой.

[14] Тем самым базовый тип обязан быть составным типом.

[15] Структурная эквивалентность не учитывает имена полей структур и тегов объединений. Структурная эквивалентность учитывает порядок следования типов тегов в объединениях. Инородные типы эквивалентны, если совпадает их строковое представление.

Имя типа объявляемого и определяемого не должно совпадать с именем простого встроенного типа[16]. Внутри одного модуля (включая его интерфейс) допускается только эквивалентное переопределение и переобъявление типов. Допускается эквивалентное переопределение переименованных, пользовательских типов и эквивалентное переобъявление пользовательских типов с параметрами. Под переопределением (переобъявлением) типа подразумевается определение типа, для которого существует предыдущее определение (объявление) типа в том же модуле (или его интерфейсе) с тем же именем. Под эквивалентным переопределением переименованного типа понимается определение переименованного типа с базовым типом, эквивалентным базовому типу предыдущего определения типа. Под эквивалентным переопределением пользовательского типа понимается определение пользовательского типа с базовым типом, эквивалентным базовому типу предыдущего определения типа. Под эквивалентным переобъявлением пользовательского типа с параметрами понимается объявление пользовательского типа с параметрами с числом параметров, равным числу параметров предыдущего объявления типа, и базовым типом, эквивалентным базовому типу предыдущего объявления типа с точностью до имён соответствующих параметров.

```
// эквивалентное переопределение переименованного типа
type complex_record = record [ real, imag: real ]
type complex_record = record [ real2: real; imag2: real ]
// эквивалентное переопределение пользовательского типа
type complex := record [ real, imag: real ]
type complex := record [ real2: real; imag2: real ]
// эквивалентное переопределение пользовательского типа с параметрами
type somerec[T1, T2, T3] := record [ a:T1; b:T2; c:T3 ]
type somerec[T3, T2, T1] := record [ a:T3; b:T2; c:T1 ]
type somerec[B, A, C] := record [ a:B; b:A; c:C ]
```

2.2. Объявления функций

Объявление функции выглядит как «*function* имя функции (список формальных параметров *returns* список типов возвращаемых значений)», где необязательные имена формальных параметров игнорируются компилятором[17]. Формой функции называется список типов её формальных параметров. Формой возвращаемых значений функции называется список типов её возвращаемых значений. Две формы функции равны, если число их типов

[16] Простые встроенные типы описываются в разделе 4.1.

[17] На самом деле, в случае несоответствия имени формального параметра в объявлении и определении функции (операции и редукции), выдаётся предупреждение.

совпадает, а соответствующие типы эквивалентны между собой. Функциями, неоднозначными по форме возвращаемых значений, называются функции, у которых совпадают имя, формы формальных параметров и различные формы возвращаемых значений.

```
// функция решения квадратного уравнения A*x^2+B*x+C
function solve_sqr_eq(A: real, B: real, C: real returns real, real)
// функции, возвращающие модуль целого и вещественного значений
function abs(integer returns integer)
function abs(real returns real)
// функции, возвращающие наибольшее из целых и вещественных значений
function max(integer, integer returns integer)
function max(real, real returns real)
function max(integer, integer, integer returns integer)
function max(real, real, real returns real)
// функции, возвращающие максимальное целое и вещественное значения
function max(returns integer)
function max(returns real)
// переобъявления функции abs
function abs(i1:integer returns integer)
function abs(i2:integer returns integer)
```

2.3. Объявления операций

Объявление операции выглядит как «*operation* знак операции (список формальных параметров *returns* список типов возвращаемых значений)», где число и типы формальных параметров и возвращаемых значений взаимозависимы со знаком операции и определяются в табл. 1 .

```
// допустимые объявления операций
operation : (complex returns integer)
operation < (complex, complex returns boolean)
operation . imag (complex returns real)
// недопустимые объявления операций
operation : (real returns integer)
operation < (complex, complex returns integer)
operation . imag (complex returns real, real)
```

Знак операции «:» задаёт операцию явного преобразования типов: «A_1 : R_1», где R_1 определяет тип, к которому приводится значение типа A_1. Тип R_1 разрешается заключать в квадратные скобки для устранения неоднозначной трактовки постфиксной операции точки и точки в имени типа, в случае неоднозначности трактуемой в пользу точки в имени типа.

Таблица 1

Допустимые объявления операций и их ограничения

<table>
<tr><th colspan="5">Знак операции</th><th>Вид операции</th><th>Ограничение</th></tr>
<tr><td colspan="5">:</td><td rowspan="2">[A_1 returns R_1]</td><td>Тип A_1 или тип R_1 должен быть пользовательским или инородным типом.</td></tr>
<tr><td colspan="5">ε [18]</td><td rowspan="3">Тип A_1 должен быть пользовательским или инородным типом.</td></tr>
<tr><td>+</td><td>-</td><td>!</td><td colspan="2">. имя</td><td rowspan="2">[A_1, …, A_n returns R_1, …, R_m]</td></tr>
<tr><td colspan="5">()</td></tr>
<tr><td colspan="5">[]</td><td>[A_1, A_2 returns R_1]</td></tr>
<tr><td colspan="2">=</td><td colspan="3"><</td><td>[A_1, A_2 returns boolean]</td><td rowspan="3">Тип A_1 или тип A_2 должен быть пользовательским или инородным типом.</td></tr>
<tr><td colspan="2">+</td><td colspan="3">-</td><td rowspan="2">[A_1, A_2 returns R_1]</td></tr>
<tr><td>*</td><td>/</td><td>%</td><td colspan="2">**</td></tr>
<tr><td>^</td><td>&</td><td>|</td><td colspan="2">||</td><td></td></tr>
</table>

```
interface complex_mod
  type complex := …
  operation : (integer returns complex)
end interface
module foo
  …
  // приведение integer типа к типу complex_mod . complex
  1 : complex_mod . complex
  …
  // то же, что и раньше, плюс взятие мнимой части
  1 : [ complex ] . imag
  …
end module
```

Знак операции «ε» задаёт неявное преобразование значения типа A_1 в значение тип R_1. Знак операции «!» задаёт унарную префиксную операцию: «! A_1». Операция со знаком «+» или «-» задаёт, в зависимости от количества формальных параметров, унарную префиксную операцию «знак операции A_1» или бинарную инфиксную операцию: «A_1 знак операции A_2». Знак операции «. имя» задаёт операцию «A_1 . имя». Знак операции «()» задаёт операцию «вызова функции»: «A_1 (A_2, …, A_n)». Знак операции «[]» задаёт операцию «доступа к элементу массива»: «A_1 [A_2]». Допустимо указывать последовательность индексов «A_1 [A_2, …, A_n]», которая разворачивается в последовательность применения операции «[]»: «A_1 [A_2] … [A_n]».

[18] Символ ε обозначает пустую цепочку, то есть объявление этой операции выглядит как «*operation* [A_1 *returns* R_1]».

```
type some1 := …
type some2 := …
operation [] (some1, integer returns some2)
operation [] (some2, real returns some2)
…
// данная операция допустима для значения A типа some1
A [ 1, 1.0, 2.0, 3.0 ]
```

Операция со знаком «=», «<», «*», «/», «%», «**», «^», «&», «|» или «||» задаёт бинарную инфиксную операцию: «A_1 знак операции A_2». Операция со знаком «=» через своё отрицание неявно задаёт операцию со знаком «!=». Операция со знаком «<» через своё отрицание неявно задаёт операцию со знаком «>=». Если одновременно заданы операции со знаками «=» и «<», то неявно определяются операции со знаками «>» и «<=».

В модуле нельзя объявлять две операции с одинаковым знаком, одинаковой формой и разной формой возвращаемых значений. В форму операций со знаками «:» и «ε» входит возвращаемый тип и, соответственно, форма возвращаемых значений этих операций пуста.

```
operation +(complex, complex returns complex)
// недопустимое объявление операции
operation +(complex, complex returns complex_record)
operation (complex returns integer)
// допустимое объявление операции
operation (complex returns real)
```

2.4. Определение контрактов

Контрактом называется набор процедур, в которых типы и параметры пользовательских типов с параметрами формальных параметров и возвращаемых значений могут быть заданы с помощью имён параметров контракта. Контракт определяется как «*contract* имя контракта [список имён параметров контракта[19]] объявления процедур *end contract*». Контракт не должен быть противоречив. Под противоречивым контрактом подразумевается контракт, в котором существует переопределение процедур в предположении, что имя параметра контракта задаёт тип, неэквивалентный другим типам.

```
contract need_add[T] // допустимый контракт
operation + (T, T returns T)
operation + (integer, integer returns integer)
function add (T, T, T returns T)
end contract
```

[19] Имена параметров должны быть различны.

```
contract any[T] // допустимый контракт без требований
end contract

contract need_add2[T] // недопустимый контракт
operation + (T, T returns T)
operation + (T, T returns integer)
end contract
```

Контракт может наследовать содержимое другого контракта, называемого базовым контрактом относительно определяемого контракта, следующим образом: «*contract* имя контракта [список имён параметров контракта] *of* имя базового контракта [список имён параметров базового контракта] объявления процедур *end contract*», где список имён параметров базового контракта входит в список имён параметров объявляемого контракта. Из определяемого контракта можно исключать объявления процедур базового контракта, предваряя их объявления ключевым словом «*no*».

```
contract need_add_mul[T] of contract need_add[T]
operation * (T, T returns T)
end contract

contract need_mul_div[T] of contract need_mul[T]
operation / (T, T returns T)
no operation + (T, T returns T)
no function add (T, T, T returns T)
end contract
```

Контракт приписывается объявлению и определению обобщенной процедуры и в её определении задаёт всё, что можно сделать с типами, заданными именами параметров контракта. В месте вызова обобщенной процедуры требуется наличие всех объявлений процедур (не являющихся обобщенными), указанных контрактом, после подстановки реальных типов вместо имён параметров контракта. Операции над строенными типами также могут быть использованы. После подстановки реальных типов вместо имён параметров контракта контракт может становиться противоречивым.

В одном модуле не могут быть определены разные контракты с одинаковым именем.

2.5. Объявления обобщенных процедур

В интерфейсе модуля можно объявить обобщенную процедуру, указав после имени функции и знака операции имя контракта и его параметров: «*of* имя контракта [список имён параметров контракта]». В типах формальных параметров и типах возвращаемых значений обобщенной процедуры допускается использование имён параметров контракта в качестве

типов и параметров пользовательских типов с параметрами. Параметры контракта, указываемые именами, не используемыми в типах формальных параметров обобщенной функции, называются свободными параметрами обобщенной процедуры. В модуле нельзя объявлять две обобщенные процедуры с одинаковым именем (или знаком операции), одинаковой формой возвращаемых значений (для функций) и разными именами контракта.

```
// определяем тип матрицы
type matrix[T] := array [..,..] of T
// определяем операцию умножения матриц
operation * of need_add_mul[T] (matrix[T], matrix[T] returns matrix[T])
// определяем операцию «замены» элемента матрицы
function insert of any[T](matrix[T], integer, integer, T returns matrix[T])
// переобъявление операции с другим контрактом недопустимо
operation * of need_mul_div[T] (matrix[T], matrix[T] returns matrix[T])
```

В обобщенных операциях допустимо использование пользовательских типов с параметрами вместо пользовательских типов там, где это требуется знаком операции. Обобщенные операции со свободными параметрами не разрешены. В модуле нельзя объявлять две обобщенные операции с одинаковым знаком, одинаковой формой и разной формой возвращаемых значений.

```
// обобщенные функции со свободными параметрами допустимы
function get_error of any[T] (returns T)
// обобщенные операции со свободными параметрами не разрешены
operation * of need_add_mul[T,T2] (matrix[T], matrix[T] returns matrix[T2])
// объявление операции недопустимо, так как уже была объявлена
// операция с такими типами формальных параметров
operation * of need_add_mul[T] (matrix[T], matrix[T] returns matrix[real])
```

Две формы обобщенной процедуры равны, если число их типов совпадает, а соответствующие типы эквивалентны между собой после эквивалентного переименования имён параметров контракта обеих форм. Тип, задаваемый именем параметра, считается эквивалентным только типу, задаваемому тем же именем параметра контракта. Под эквивалентным переименованием имён последовательности подразумевается последовательность имён, полученная после последовательного назначения новых имён, взятых из общей последовательности имён, при сохранении равенства равных имён.

```
function some of some[T1, T2, T3, T4] ( T1, T2, T2 returns T3, integer, T4)
// объявление той же обобщенной функции
function some of some[A, B, C, D] ( D, C, C returns B, integer, A)
// объявление других обобщенных функций
function some of some[A, B, C, D] ( D, C, D returns B, integer, A)
function some of some[A, B, C, D] ( D, C, C returns B, A, integer)
```

2.6. Импорт типов и контрактов

Импортировать все типы и контракты из других интерфейсов модулей можно конструкцией «*import* список имён интерфейса модуля». Можно импортировать конкретные типы или контракты по их имени конструкцией «*import* имя интерфейса модуля: объект импорта, …, объект импорта», где объектом импорта может быть тип «*type* имя типа» или контракт «*contract* имя контракта». Ключевые слова «*type*» и «*contract*» являются обязательными в случае наличия в импортируемом модуле объявления функции или одновременного наличия типа и контракта с таким же именем.

```
interface A
  type A1 = … type A2 = … type B1 := … type B2 := …
  contract A2 … end contract
  function B1 …
end interface

interface B
  import A
  // повторный импорт не ошибочен
  import A: A1, type A2, contract A2, type B1, B2
  // неоднозначный импорт ошибочен
  import A: A2
  // неоднозначный импорт ошибочен даже,
  // если функция B1 не может быть импортирована в интерфейс модуля
  import A: B1
```

Можно импортировать все типы и контракты модуля, кроме типов и контрактов, указанных конструкцией «*import* имя интерфейса модуля - объект импорта, …, объект импорта», например:

```
interface B
  // импортирует всё содержимое интерфейса модуля,
  // за исключением контракта A2 и типа B2
  import A - contract A2, B2
```

Импортируемые типы и контракты становятся частью интерфейса модуля, как если бы они были в нём определены или объявлены вместо конструкции *import*[20], за исключением того, что:

– при выборочном импорте типов неуказанные типы не импортируются, даже если они участвуют при построении импортируемых типов и для доступа к их содержимому (типам, лежащим в их основе) необходимо их импортировать[21];

[20] Таким образом отпадает необходимость использования оператора «.» в интерфейсе модуля.

[21] Данное требование, на первый взгляд, сводит объявление импорта типа до уровня его декларации (opaque type declaration в терминологии языка Sisal 2.0), однако импорт типа под-

– импортированный в интерфейсе модуля пользовательский тип и пользовательский тип с параметрами считается эквивалентным импортируемому типу.

```
interface A
  type someint := integer
  type someint2 := array of someint
end interface

interface B
  import A: type someint2
  // тип someint2 в модулях A и B эквивалентен и может быть использован
  function foo (someint2 returns null)
  // ошибка, тип someint не был импортирован
  function foo (someint returns null)
```

3. СТРУКТУРА МОДУЛЯ

Интерфейс модуля на языке Sisal начинается с ключевого слова *module*, за которым следует его имя. Модуль может содержать определения процедур, их объявления (предваряемые ключевым словом *forward*), определения и объявления типов, определения контрактов, а также конструкции импорта содержимого интерфейсов модулей. Содержимое модуля может разделяться точкой с запятой. Все объявления и определения влияют только на последующий текст модуля. Модуль завершается ключевыми словами «*end module*».

Например, далее приводится пример модуля «math», содержащего определения функций для вычисления факториала и числа Фибоначчи:

```
module math
  function fact (n: integer returns integer) // факториал числа n
    if n = 1 then 1 else fact(n-1)*n end if
  end function
  function fib (n: integer returns integer) // число Фибоначчи с номером n
    if n = 1 | n = 2 then 1 else fib(n-1) + fib(n-2) end if
  end function
end module
```

разумевает наличие определения типа (пусть и в интерфейсе другого модуля), что позволяет контролировать эквивалентность типов в модуле определения интерфейса и в модуле его использования.

3.1. Определение процедур

Определение процедуры выглядит как объявление процедуры (называемое заголовком определения), в котором должны быть указаны имена формальных параметров, и за которым следует список выражений[22]. Размерность списка выражений равна количеству возвращаемых значений, а типы значений можно неявно преобразовать[23] в соответствующие типы возвращаемых значений. Определение функции завершается ключевыми словами «*end function*». Определение операции завершается ключевыми словами «*end operation*». Определение процедуры также является соответствующим объявлением, если его не было сделано ранее.

```
function minmax1 (a: integer, b: integer returns integer, integer)
  if a < b then a, b else b, a end if
end function

// определение функции minmax2, эквивалентной функции minmax1
function minmax2 (a: integer, b: integer returns integer, integer)
  if a < b then a else b end if,
  if a < b then b else a end if
end function

// определение функции minmax3, использующее функцию minmax2,
// объявленную её определением
function minmax3 (a: integer, b: integer returns integer, integer)
  minmax2(a, b)
end function

// пример определения операции
operation - (c: complex returns complex)
  complex (-c.real, -c.imag)
end operation
```

Определение процедуры задаёт три области видимости объявлений: N_1, N_2 и N_3. Область видимости N_1 содержит объявления процедур, объявленных ранее. Область видимости N_2 не пуста только для определений обобщенных процедур, для которых она содержит объявления процедур контракта. Область видимости N_3 содержит имена формальных параметров определения процедуры, которые должны быть различными. Выражения процедуры могут определять последующие области видимости имён. Имена из областей видимости с большим номером перекрывают имена из областей видимости с меньшим номером. Также выражения могут использовать имена типов, объявленных и определённых ранее.

[22] Смотри раздел 5 для определения понятия списка выражений.

[23] Определение встроенных неявных преобразований типов находится в разделе 4.

```
// операция, возвращающая первые два значения,
// зависящие от определения операций контракта, и значение 3.0
function foo of need_add[T] (a: T, b: T returns T, integer, real)
  a + b, // используется операция контракта «operation + (T, T returns T)»
  let a := 1; b := 2 in
    a + b, // «operation + (integer, integer returns integer)» из контракта
    a:real + b // обычное сложение вещественного и целого значений, так как
               // операция из контракта неприменима,
              // ввиду отсутствия неявного преобразования real в integer
  end let
end function
```

Если у модуля в программе существует его интерфейс, то в модуле необходимо определить все процедуры, объявленные в его интерфейсе. Также в модуле необходимо определить все объявления процедур модуля. Определение функции сопоставляется её объявлению с той же формой формальных аргументов и возвращаемых значений. Определение операции сопоставляется её объявлению с той же формой формальных аргументов. Переопределение определений процедур не допускается.

```
// объявление и определение функции в модуле
forward function one (returns integer)
function one (returns integer) 1 end function
// объявление и определение различных функций
forward function neg (real returns real)
function neg (n: real returns real) -n end function

function nothing (returns null) nil end function
// переопределение функции nothing недопустимо
function nothing (returns null) nil end function
```

3.2. Импорт содержимого интерфейсов модулей

Конструкция импорта содержимого интерфейса модуля делает возможным использовать в модуле импортируемые процедуры, типы и контракты, в нём объявленные и определенные. Импортировать всё содержимое интерфейсов модулей можно конструкцией «*import* список имён интерфейса модуля».

Можно импортировать конкретные объявления или определения конструкцией «*import* имя интерфейса модуля: объект импорта, …, объект импорта». Объектом импорта может выступать:

- тип «*type* имя типа»;
- контракт «*contract* имя контракта»;
- конструкция «*function* имя функции [..]» для импорта всех объявлений функций с указанным именем;

- конструкция «*function* имя функции [список типов формальных параметров]» для импорта конкретной функции, однозначной по возвращаемым значениям;
- конструкция «*function* имя функции [список типов формальных параметров *returns* список типов возвращаемых значений]» для импорта конкретной функции;
- конструкция «*function* имя функции [список типов формальных параметров инородной функции]» для импорта инородной процедуры (инородная операция указывается её именем функции), где типы формальных параметров списка формальных параметров функции могут предваряться ключевыми словами «*raw*», «*in*», «*out*» и «*in out*»;
- конструкция «*function* имя функции *of* [имена параметров контракта] [список типов формальных параметров]» для импорта конкретной обобщённой функции, однозначной по возвращаемым значениям, где список типов формальных параметров зависит от имён указанных параметров контракта так же, как и у обобщенной функции, которую нужно указать;
- конструкция «*function* имя функции *of* [имена параметров контракта] [список типов формальных параметров *returns* список типов возвращаемых значений]» для импорта конкретной обобщённой функции;
- конструкция «*operation* знак операции [тип формального параметра *returns* тип возвращаемого значения]» для импорта операций со знаками «:» и «ε» и конструкция «*operation* знак операции [список типов формальных параметров]» для импорта операций с другими знаками;
- конструкция «*operation* знак операции *of* [имена параметров контракта] [тип формального параметра *returns* тип возвращаемого значения]» для импорта обобщённых операций со знаками «:» и «ε» и конструкция «*operation* знак операции *of* [имена параметров контракта] (список типов формальных параметров)» для импорта обобщённых операций с другими знаками.

Ключевые слова *type*, *contract* и *function* не обязательны, если в импортируемом модуле существуют только тип, контракт или функции с указанным именем. Ключевое слово *function* не обязательно, если после имени функции указана форма функции.

```
interface A
  type A1 = integer
  function A1 (integer returns integer)
  function A1 (integer returns integer, integer)
  function A1 (real returns integer)
  function A1 (integer, integer returns integer)
  function A2 (integer returns integer)
  function A2 (real returns real)
  contract any[T] end contract
  function A2 of any[T1, T2] (array of T1, T2 returns stream of T1)
  function A2 of any[T1, T2] (array of T1, T2 returns array of T1)
  function A2 of any[T1, T2] (array of T1, T1 returns stream of T1)
  operation (complex returns integer)
  contract need_intcast[T] operation (T returns integer) end contract
  operation of need_intcast[T] (matrix[T] returns matrix[integer])
  operation < (complex, complex returns boolean)
  contract need_cmp[T] operation < (T, T returns boolean) end contract
  operation < of need_cmp[T] (matrix[T], matrix[T] returns boolean)
end interface
module B
  // импортировать все объявления интерфейса модуля A
  import A
  // импортировать тип A1, контракт any и все функции с именами A1 и A2
  import A: type A1, any, function A1, A2
  // импортировать конкретные функции,
  // однозначные по возвращаемым значениям
  import A: A1 [real], function A1[integer], A1 [integer, integer]
  // импортировать конкретную функцию,
  // не обязательно однозначную по возвращаемым значениям
  import A: A1 [integer returns integer], A1 [integer returns integer]
  // импортировать обобщенную функцию,
  // однозначную по возвращаемым значениям (контракт any не  импортируется)
  import A: A2 of [T1, T2] [array of T1, T1]
  // импортировать обобщенную функцию,
  // не обязательно однозначную по возвращаемым значениям
  import A: A2 of [T1, T2] [array of T1, T2 returns stream of T1]
  // импортировать операции неявного преобразования типов
  import A: operation [complex returns integer]
  import A: operation of [T] [matrix[T] returns matrix[integer]]
  // импортировать операции сравнения
  import A: operation < [complex, complex]
  import A: operation < of [X] [complex[X], complex[X]]
```

Можно импортировать всё содержимое модуля, за исключением указанных объявлений и определений, конструкцией «*import* имя интерфейса модуля - объект импорта, …, объект импорта».

```
module B
  // импортировать всё, кроме функций с именем A2 и конкретной операции
  import A - A2, operation [complex returns integer], A2[integer]
```

При возникновении неоднозначности имени функции, типа или контракта, возникающей при наличии более одной функции, более одного типа или более одного контракта с одинаковым именем, принадлежащего разным модулям, перед неоднозначным именем обязательно указание имени модуля «имя модуля .»[24], которому принадлежит функция, тип или контракт. В одном модуле не может быть более одной операции с одинаковым знаком и формой[25].

```
interface A type T := integer end interface
interface B
  import A: type T
  function add (T, T returns T)
  operation + (T, T returns T)
end interface
interface C
  import A: T
  function add (T, T returns T)
  operation + (T, T returns T)
end interface
module D
  import A, B // импортировать интерфейс модуля A необязательно
  // импортировать интерфейс модуля C нельзя, так как из интерфейса модуля
  // B уже была импортирована операция «operation + (T, T returns T)»
  import C
  // эти предложения импорта правильные
  import C - operation + [T, T]
  import C: add [T, T]
  // в модуле D доступны две функции add: B.add и C.add
```

Если в программе существует интерфейс модуля **A**, то, неявно, его содержимое целиком импортируется в модуль **A**, как если бы конструкция импорта «*import* **A**» находилась сразу же после «*module* **A**».

```
interface some
  function foo(integer returns integer)
end interface
module some
  // функция foo была импортирована в модуль some неявным образом
  function foobar (i: integer returns integer) foo(i) end function
end module
```

[24] Данный префикс можно использовать не только для устранения неоднозначности имён.
[25] Таким образом, не возникает неоднозначности использования знака операции.

4. ТИПЫ И ОПЕРАЦИИ НАД НИМИ

Язык содержит следующие встроенные простые[26] типы: пустой (null), булевский (boolean), символьный (character), целый (integer) и вещественный (real). Встроенные простые типы не эквивалентны между собой. К встроенным составным типам[27] относятся потоки (*stream*), массивы (*array*), записи (*record*), объединения (*union*) и функции (*function*). Тип унарного выражения можно задать следующим образом: «*type* (унарное выражение)», например, «*type* (1.0+2)» равняется типу real.

Для каждого типа, кроме типа null и возможно инородных типов, существует выделенное ошибочное значение. Если не оговорено обратное, и аргумент операций над встроенными типами или аргумент предопределённых функций ошибочен, то результаты тоже будут ошибочными значениями. Узнать, ошибочно ли значение выражения, можно с помощью постфиксной операции «(выражение) *is error*».

4.1. Простые типы

Пустой тип состоит из единственного значения *nil*[28]. Никаких операций для пустого типа не определено.

Булевский тип состоит из значений истины (*true*), лжи (*false*) и ошибочного значения (*error*[boolean]). Для булевского типа определены бинарные[29] операции равенства («=»), неравенства («!=»), конъюнкции («&»), дизъюнкции («|»), исключающей дизъюнкции («^») и унарная[30] операция отрицания («!»). Определены операции явного преобразования между значениями булевского типа и значениями целого типа, так что значение *true* соответствует ненулевому числу, а значение *false* соответствует нулю.

```
// данные выражения истинны
! false, false = false, true = true, true != false, false != true,
true&true, true|false, false|true, true|true, true^false, false^true,
1 : boolean, true : integer = 1, false : integer = 0,
(error[boolean] = error[boolean]) is error

// данные выражения ложны
! true, true = false, false = true, true != true, false != false,
true&false, false&true, false&false, false|false, false^false, true^true,
0 : boolean
```

[26] Простыми называются типы, не определяемые через другие типы.
[27] Составными называются типы, определяемые через другие типы.
[28] Пустой тип используется для тегов объединения с неуказанными типами.
[29] Бинарные операции имеют два аргумента.
[30] Унарные операции имеют один аргумент.

Символьный тип содержит как минимум символы стандарта ASCII [12] и ошибочное значение (*error*[character]). Каждому символу соответствует уникальный неотрицательный целый код (для символов ASCII код описывается в соответствующем стандарте). Символы упорядочены в соответствии с порядком их кодов. Для символьного типа определены бинарные операции равенства, неравенства, больше («>»), меньше («<»), больше или равно («>=») и меньше или равно («<=»). Операция разности («-») двух символьных значений возвращает целое значение разности кодов символов.

```
// данные выражения истинны
'S' - 'S' = 0, '0' < '1' < '9' < 'A' < 'B' < 'Z' < 'a' < 'b' < 'z'
```

Определены операции явного преобразования между значениями символьного типа и значениями целого типа, так что символ соответствует числу кода этого символа. Литералы символьного типа имеют вид знака символа в одинарных кавычках. Также допускаются литералы, приведенные в табл. 2 , где ASCII-коды символов указаны как целые литералы языка Sisal.

```
'S', '\x53', '\83', '\o123' // все способы задания символа S
```

Таблица 2

Коды обратной косой черты (escape-последовательности)

Литерал	'\"'	'\\'	'\a'	'\b'	'\f'	'\n'	'\r'	'\t'	'\v'
Код	16#27	16#5C	16#7	16#8	16#C	16#A	16#D	16#9	16#B
Литерал	'\10-ричный код D'			'\o8-ричный код O'			'\h16-ричный код H'		
Код	10#10-ричный код D			8#8-ричный код O			16#16-ричный код H		

Целый тип содержит все числа достаточные для представления кодов символов символьного типа, их отрицания и ошибочное значение (*error*[integer]). Значения типа задаются цепочкой десятичных чисел, для повышения читаемости которой везде, кроме ее начала, могут использоваться символы подчеркивания. Знак числа, как и для вещественных чисел, считается унарной операцией, и поэтому между ним и числом допускается произвольное число пробельных символов. Целые литералы могут задаваться в произвольной системе счисления: «основание#число», где ее основание является числом от 2 до 36.

```
2#100_0000 = 8#100 = 16#40 = 36#1S = 64 // данное выражение истинно
```

Вещественный тип содержит значения, определяемые стандартом IEEE-754 [13], и ошибочное значение (*error*[real]). Значения вещественных типов

задаются десятичными литералами, отличающимися от целых литералов наличием точки и/или знаком экспоненты. Также для вещественных чисел символы подчеркивания не могут идти после десятичной точки. Вещественное число задается литералом простой или экспоненциальной формы. Простая форма является десятичным числом, разделенным знаком точки на две части, одна из которых может быть пуста. Экспоненциальная форма состоит из двух частей, разделенных буквой «e» или «E». Левая часть — это десятичное число, возможно разделяемое на две части точкой, которая может стоять вначале. Правая часть — это необязательный знак минуса или плюса и десятичное число.

```
5.0 = 5. = 0.5e1 = .5E1 = 50e-1 = 50_000.000E-4 // данное выражение истинно
```

Целый и вещественный типы имеют общее название числовых типов. Для числовых типов определены унарные операции идентичности («+»), смены знака («-»). Для числовых типов определены бинарные операции сложения («+»), вычитания («-»), умножения («*»), деления («/»), возведения в степень («**»), равенства, неравенства, больше, меньше, больше или равно и меньше или равно. Для целых значений определена бинарная операция взятия остатка от деления или модуля («%»). Определена операция неявного (и явного) преобразования целого значения в значение вещественного типа. Определена операция явного преобразования вещественного значения в значение целого типа «X : integer», возвращающая значение «floor (0.5 + X)».

```
// данные выражения истинны
+1 = 1, -1 = 0-1, 2+2 = 4, 2*3 = 6, 5/2 = 2, 5%2 = 1, 2**5 = 32,
+1.0 = 1.0, -1.0 = 0.0-1.0, 2.0*3.0 = 6.0, 5.0/2.0 = 2.5, 2.0**5.0 = 32.0,
1_000_000_000 : real = 1_000_000_000.0, // вероятна потеря точности
2.49 : integer = 2, 2.5 : integer = 3, 2.51 : integer = 3
```

Операция над целыми значениями порождает целое значение. При делении и возведении в степень целых значений берется целая часть результата. Операция над вещественными значениями порождает вещественное значение. Если операция над целыми значениями или операция явного преобразования вещественного значения в целое значение порождает число, не представимое типом integer, то порождается значение «*error*[integer]». Если в результате выполнения операции деления или модуля от целых операндов происходит деление на ноль, то возвращается значение «*error*[integer]». Для вещественного деления на ноль возвращается значение $\pm\infty$ стандарта IEEE-754.

```
// данные выражения истинны
2.0 + 2 = 4.0, 2.0:integer + 2 = 4, (1 / 0) is error, (1 % 0) is error
```

В интерфейсе модуля «std» определяются следующие функции:

```
// возвращает наибольшее целое число, не большее, чем вещественный аргумент
function floor (real returns integer)
// возвращает целое число, равное вещественному аргументу без дробной части
function trunc (real returns integer)
// возвращает модуль числа
function abs (integer returns integer)
function abs (real returns real)
// возвращает минимум двух чисел
function min (integer, integer returns integer)
function min (real, real returns real)
// возвращает максимум двух чисел
function max (integer, integer returns integer)
function max (real, real returns real)
```

4.2. Потоки

Тип потока описывается как «*stream of* тип элемента потока» и содержит возможно бесконечные цепочки последовательно доступных элементов одного типа и ошибочное значение «*error* [*stream of* тип элемента потока]». Типы потоков эквивалентны, если эквивалентны типы их элементов.

```
type si1 = stream of integer
// тип si1 эквивалентен типу si2
type integer2 = integer
type si2 = stream of integer2
```

Поток можно сконструировать в циклическом выражении[31] или с помощью выражения конструктора потока «*stream of* тип элемента потока [список значений элементов потока]» или «*stream* тип потока [список значений элементов потока]». Список значений элементов потока может отсутствовать для пустого потока. Для непустого списка значений элементов потока тип потока может отсутствовать и неявно задаваться типом первого элемента потока. Каждый элемент списка значений элемента потока последовательно задаёт одно или несколько значений элементов потока, начиная с элемента с индексом 1, и может быть выражением[32] или триплетом. Триплетом является структура вида «нижняя граница .. верхняя граница .. шаг», где все три элемента должны быть унарными выражениями целого типа и часто могут быть опущены[33] независимо друг от друга (шаг опускается вместе с точками «..», идущими впереди).

[31] Циклические выражения рассматриваются в разделе 5.5.
[32] Выражение задаёт количество элементов потока, равное его размерности.
[33] Триплеты используются не только в выражениях конструктора потока.

Триплет выражения конструктора потока задаёт арифметическую прогрессию элементов с указанной нижней границей и шагом, которые меньше или равны указанной верхней границы. Триплет задаёт по крайней мере один элемент (равный нижней границе). Если опущена нижняя граница, то она предполагается равной единице. Если опущена верхняя граница, то она предполагается равной бесконечности. Верхняя граница может быть опущена только в триплете, который является последним элементом списка значений элементов потока. Если опущен шаг, то он предполагается равным единице или минус единице, если нижняя граница больше верхней границы. Выражения, задающие конструкторы потоков, приведены ниже:

```
stream of integer [1..], // 1, 2, 3, …
stream si1 [1..4], // 1, 2, 3, 4
stream si2 [10..1..-3], // 10, 7, 4, 1
stream [..], // такой же поток, как и в первом выражении
stream [1..-3..1], stream of integer2 [], // пустой поток целых чисел
stream [], // пустой поток недопустим без указания его типа
stream of integer [0 .. .. -1], // 0, -1, -2, …
// поток вещественных чисел N-1, 1.0, 2.0, 3.0, 0.0, 0.0, 0.0
stream [(N-1):real, 1..3, 0, 0, 0],
// поток целых чисел N-1, 1, 2, 3, 0, 0, 0 (если N является целым числом)
stream [N-1, 1..3, 0, 0, 0]
```

Поток поддерживает выражение выбора элементов, задаваемое как «поток [выбирающее выражение]», где выбирающее выражение может быть унарным выражением целого типа (индексом), триплетом или унарным выражением типа целого потока или массива (индексным вектором). Выражение выбора, задаваемое индексом, возвращает выбираемый элемент. Выборка, задаваемая триплетом или индексным вектором, возвращает поток, составленный из выбираемых элементов. Если элемента с указанным индексом в потоке нет, то из потока выбирается ошибочное значение «*error* [тип элемента потока]». Все выражения выбора элементов потока являются более удобной записью совокупности выражений «поток [1]» (функция first) и «поток [2 ..]» (функция rest).

```
// выражение возвращает значения:
// stream [50, 60, 70, 80, 90, 100], 40, stream [70, 90] и 80
let  S := stream of integer [40, 50, 60, 70, 80, 90, 100 ]; N := 7
in S[2..], S[1], S[4..N-1..2], S[N-2]
end let,
// выражение S[4] являет сокращенной записью выражения, лежащего ниже:
let TS1 := S[2..]; TS2 := TS1[2..]; TS3 := TS2[2..] in TS3[1] end let
// выражение S[3..4] являет сокращенной записью выражения, лежащего ниже:
let TS1 := S[2..]; TS2 := TS1[2..]; V3 := TS2[1];
    TS3 := TS2[2..]; V4 := TS3[1]
in stream [V3, V4] end let
```

Для двух потоков определена бинарная операция конкатенации («||»), возвращающая поток, склеенный из двух указанных потоков. Программа неправильна, если типы потоков неэквивалентны, и не существует подходящей операции неявного преобразования типов элементов потоков[34].

```
// данное выражение истинно
let S1 := stream [10]; S2 := stream [20, 30];
    S3 := stream [40]; S4 := stream of integer []
in S1 || S2 || S3 || S4 end let = stream [10, 20, 30, 40]
```

Если тип элемента потока допускает префиксные и постфиксные[35] операции, то они допустимы и для этого потока, порождая поток с элементами, полученными после поэлементного применения операции над значениями исходного потока. Недопустимы постфиксные операции вызова функции более чем с одним результатом.

Если типы элементов двух потоков допускают инфиксные операции, то они допустимы и для этих потоков, порождая поток с элементами, полученными при поэлементном применении операции. Элементы меньшего по размеру потока дополняются ошибочными значениями.

Если тип элемента потока и значения, не являющегося потоком, допускают инфиксные операции, то они допустимы и для этого значения и потока, порождая поток с элементами, полученными при поэлементном применении операции для исходного потока и значения.

```
// выражение возвращает stream [false, true], stream [5, 7, 9],
// stream [8.0, 10.0, 12.0] и stream [1, 4, 27]
let S1 := stream [1, 2, 3]; S2 := stream [4, 5, 6];
    S3 := stream [true, false]
in !S3, S1 + S2, S2 * 2.0, S1 ** S1 end let
```

В интерфейсе модуля «std» определяются следующие функции и операции:

```
contract any[T] end contract // на тип T нет никаких ограничений
// возвращает true, если поток пуст, и false иначе
function empty of any[T] (stream of T returns boolean)
// возвращает поток, склеенный из двух указанных потоков
operation || of any[T] (stream of T, stream of T returns stream of T)
// следующие функции применяют соответствующие операции поэлементно
function floor (stream of real returns stream of integer)
function trunc (stream of real returns stream of integer)
function abs (stream of integer returns stream of integer)
```

[34] Сначала проверяется возможность неявного преобразования типа элемента второго потока к типу элемента первого потока, а потом возможность обратного преобразования.

[35] Кроме постфиксной операции квадратных скобок («[]»), так как она конфликтует с выражением выбора элементов потока.

```
function abs (stream of real returns stream of real)
function min (stream of integer, stream of integer
              returns stream of integer)
function min (stream of real, stream of real returns stream of real)
function max (stream of integer, stream of integer
              returns stream of integer)
function max (stream of real, stream of real returns stream of real)
```

4.3. Массивы

Тип массива описывается как «*array* форма массива *of* тип элемента массива» и содержит конечные цепочки элементов одного типа с прямым доступом по их многомерному индексу и ошибочное значение «*error* [*array* форма массива *of* тип элемента массива]». Форма может иметь свободный «[список двойных точек]» или фиксированный вид «[список дуплетов]». Дуплетом является структура вида «нижняя граница .. верхняя граница», где нижняя и верхняя границы являются унарными выражениями целого типа, чьи значения известны во время трансляции текущего модуля[36]. Нижняя граница может быть опущена и по умолчанию полагается равной единице. Верхняя граница должна быть больше или равна нижней границе. Форма может быть опущена и по умолчанию полагается равной «[..]». Количество размерностей массива задаётся размерностью формы массива, равной количеству элементов её списка двойных точек или дуплетов.

```
type Arr1 = array of integer // одномерный массив целых чисел
type Arr2 = array [..] of integer // одномерный массив целых чисел
type Arr3 = array [..,..] of integer // двухмерный массив целых чисел
type Arr4 = array [1..4] of integer // одномерный массив 4-х целых чисел
type Arr5 = array [1..5] of integer // одномерный массив пяти целых чисел
type Arr6 = array [..2,..3] of integer // 2x3 массив целых чисел
type Arr7 = array [2..5] of integer // одномерный массив 4-х целых чисел
type Arr8 = array [..3,..2] of integer // 3x2 массив целых чисел
```

Два типа массива эквивалентны, если эквивалентны типы их элементов и массивы имеют одинаковую форму. Формы массивов совпадают, если они имеют одинаковую размерность, они обе свободны или фиксированы, и у фиксированных форм совпадает количество элементов каждой размерности.

[36] Значение нижней и верхней границ массива не должно быть ошибочным значением.

```
// массив типа Arr1 эквивалентен массиву типа Arr2
// массив типа Arr4 эквивалентен массиву типа Arr7
// более никаких эквивалентностей среди типов массивов Arr1, …, Arr8 нет
```

Строковые литералы рассматриваются как одномерные массивы символов «*array of* character» (в интерфейсе модуля «std» находится определение типа «*type* string = *array of* character») с единичной нижней границей и задаются цепочкой символов, заключенной в двойные кавычки. Для задания символов строки допустимы все обозначения таблицы 2. Единственная особенность связана с синтаксисом задания символа его кодом: если за ним находится символ точки с запятой, то он считается маркером окончания кода и к строке не добавляется[37]. Если непосредственно перед начальной кавычкой находится символ «@», то все специальные обозначения символов, кроме цепочки «\"», воспринимаются буквально. Последовательные строковые литералы, возможно расположенные на разных строках, склеиваются в один.

```
"\83isal", "\83;isal", // строковой литерал "Sisal"
"foo" "bar", // один строковой литерал "foobar"
"foo" @"b\ar", // один строковой литерал "foob\\ar"
@"\foo" "bar", // один строковой литерал "\\foobar"
```

4.3.1. Выражение конструирования массива

Массив можно сконструировать в циклическом выражении или с помощью выражения конструктора массива. Выражение конструктора массива имеет вид «*array of* тип элемента массива [список значений элементов массива]», «*array* форма фиксированного вида *of* тип элемента массива [значения элементов массива]», «*array* форма фиксированного вида тип массива [значения элементов массива]» или «*array* тип массива с формой фиксированного вида [значения элементов массива]».

В указываемой форме необязательно использовать выражения, вычисляемые во время трансляции. В указываемой форме разрешается указывать перед дуплетом его имя «имя *in*», которое является уникальным для формы. Областью действий указанных имён является область значений элементов массива. Указываемая форма должна иметь количество размерностей, равное размерности указанного типа массива. В указываемой форме для типа массива с фиксированной формой можно указывать только имена нужных дуплетов, ставя двоеточие «..» для размерностей с не указанными именами.

[37] Тем самым возможно отделять код символа от идущих следом символов, которые могут быть проинтерпретированы как его часть.

Если форма конструируемого массива неизвестна, то конструируется одномерный массив с нижней границей равной единице и список значений элементов массива задаётся полностью аналогично конструктору потока. Для пустого массива список значений элементов массива может отсутствовать. Для непустого массива тип элемента массива может отсутствовать, возможно, вместе с ключевыми словами «*array of*» (для первого случая) и неявно задаваться типом первого указанного элемента массива.

```
// задаёт три одинаковых массива
array of integer [1, 2, 3], array of [1, 2, 3], [1, 2, 3]
```

Если форма массива задана, то значения элементов массива задаются непосредственно как «:= список значений элементов массива в row-major порядке»[38] или как список расположенных элементов, разделяемых точкой с запятой. Значения элементов массива задаются значениями типа **T**, эквивалентного типу элементов массива или неявно к нему приводимого.

```
// одинаковые одномерные массивы
array [1..3] of integer [:= 1, 2, 3], array [1..3] of [:= 1, 2, 3],
// одинаковые двухмерные массивы [[1, 2, 3], [4, 5, 6]]
array [1..2, 1..3] of integer [:= 1, 2, 3, 4, 5, 6],
array [..2, ..3] of [:= 1, 2, 3, 4, 5, 6],
// такой же массив, но с фиксированной формой
array array [1..2, 1..3] of integer [:= 1, 2, 3, 4, 5, 6]
```

Расположенные элементы задаются как «положение := расположенные значения элементов массива» или «*else* := расположенные значения элементов массива». Положение указывает задаваемые элементы массива и задается как список выражений целого типа или возможно именованных триплетов и индексных векторов. Если компонент положения ошибочен, или положение задаёт элементы вне массива, то указанные значения элементов массива не попадают в массив. Все незаданные элементы массива равны ошибочным значениям. Элементы, расположенные с помощью ключевого слова «*else*» (которое должно являться последним элементом списка расположенных элементов), соответствуют одному или более не заданных элементов массива[39]. Если в выражении конструктора массива описание расположения элементов пересекается, то возвращается ошибочный массив.

[38] При row-major порядке перечисления элементов массива элементы перечисляются, начиная с внутренней размерности массива. Далее именно такой порядок подразумевается в последовательности всех элементов массива.

[39] В выражении конструктора массива не может быть более одного описания расположения элементов с ключевым словом «*else*», и это описание не может быть единственным.

```
// массив [1, 2, 3]
array [1..3] of [1 := 1; 2 := 2; 3 := 3; 4 := 4],
array [1..3] of [1 := 1; 2 := 2; else := 3],
// массив array of real [1.0, 2.0, 3.0]
array [1..3] of [2 := 2.0; 1 := 1; 3 := 3],
// массив [[1, 0, 3], [4, 0, 6]]
array [1..2, 1..3] of [1,1 := 1; 1,3 := 3; 2,1 := 4; 2,3 := 6; else := 0],
// незаданные элементы массива равны ошибочным значениям
array [1..3] of integer [:= 1, 2], // массив [1, 2, error[integer]]
array [1..3] of [2 := 1; 3 := 2], // массив [error[integer], 1, 2]
array [1..3] of [1 := 1; 2 := 2; 3 := 3; 1 := 1] // ошибочный массив
```

Триплеты и индексные вектора могут разделяться ключевым словом «*dot*», а не запятой, и их последовательность называется последовательностью *dot*-элементов. Область действия имён триплетов и индексных векторов распространяется на последующие элементы списка выражений и расположенные значения элементов массива, если они заданы одним унарным выражением типа **T**.

Размерность списка положения должна быть равна размерности массива. Выражение задаёт одну или несколько размерностей списка в зависимости от своей размерности. Триплет и индексный вектор задают одну размерность списка. Каждая размерность списка положения задаёт индексы элементов, выбираемых из этой размерности. Неуказанная нижняя граница триплета равняется нижней границе соответствующей размерности массива. Неуказанная верхняя граница триплета равняется верхней границе соответствующей размерности массива. Неуказанный шаг триплета равняется единице.

Размерность формы положения определяется как размерность массива минус количество триплетов и индексных векторов плюс количество ключевых слов «*dot*». Каждой размерности формы положения соответствует триплет, индексный вектор или последовательность *dot*-элементов. Каждая размерность формы положения имеет нижнюю границу, равную нижней границе соответствующей размерности, и верхнюю границу, равную сумме нижней границы и количества выбираемых элементов соответствующей размерности. Последовательность *dot*-элементов имеет верхнюю границу, равную наибольшей размерности объединяемых триплетов и индексных векторов. Индексы последовательности *dot*-элементов изменяются вместе до тех пор, пока не исчерпаются индексы последнего элемента. Закончившиеся индексы равны значению «*error*[integer]».

Расположенные значения элементов массива могут задаваться непосредственно их перечислением в row-major порядке, унарным выражением типа **T** или массивом с размерностью формы положения и элементами типа **T**. Унарное выражение типа **T** задаёт все элементы, указанные положением.

Из массива с размерностью формы положения и элементами типа **T** выбираются элементы с индексами формы положения.

```
array [1..3] of [1..3 := 1, 2, 3], // массив[[1, 2, 3]
array [..2, ..3] of [..,.. := 1], // массив [[1, 1, 1], [1, 1, 1]]
// массив [[1, 2, 3], [4, 5, 6]]
array [..2, ..3] of [1,.. := 1, 2, 3; 2,.. := 4, 5, 6],
// массив [[1, 2, 3], [4, 5, 6]]
array [..2, ..3] of [1,.. := [1, 2, 3]; 2,.. := [4, 5, 6]],
// массив [[2, 0, 0], [0, 4, 0], [0, 0, 6]]
array [..3, ..3] of [i in .. dot j in .. := i+j; else := 0],
// массив [[1, 0, 0], [0, 2, 0], [0, 0, 3]]
array [..3, ..3] of [.. dot .. := 1, 2, 3; else := 0],
// массив [[1, 1, 1], [0, 1, 1], [0, 0, 1]]
array [..3, ..3] of [i in .., i .. := 1; else := 0],
// массив [[0, 1, 2], [4, 5, 0]]
array [..2, ..3] of [1, [3, 2] := 1, 2; 2, [1, 2]. := 4, 5; else := 0]
```

4.3.2. Операции над массивами

Массив поддерживает выражение выбора и замещения элементов. Выражение выбора элементов массива имеет вид «массив [положение]», где положение описывалось в разделе 4.3.1. Выражение выбора элементов массива возвращает значение элемента массива, если размерность формы положения равна нулю. Если размерность формы положения не равна нулю, и количество элементов каждой размерности не зависит от других размерностей (форма массива задаёт прямоугольный массив), то возвращается массив с формой положения. Если размерность формы положения не равна нулю, и форма массива не задаёт прямоугольный массив, то возвращается массив массивов, каждый из которых соответствует последовательности размерностей формы, задающей прямоугольный массив.

```
// выражение let вычисляет значения:
// 3.0, [5, 6], [1.0, 4.0, 2.0, 1.0], true и true
let A := array [1..4] of real [1.0, 2.0, 3.0, 4.0];
    B := array [1..3, 2..4] of integer [1,.. := 1, 2, 3;
                                        2,.. := 4, 5, 6;
                                        3,.. := 7, 8, 9 ];
    U := array of integer [1, 4, 2, 1]
in A[3], B[2, 3..4], A[U],
   B [1..3 dot 4..2..-1] = array [3, 5, 7],
   B [3..2..-1 dot 2..4..2] = array [7, 6]
end let
```

Выражение замещения элементов массива имеет вид «массив [список расположенных элементов[40]]» и возвращает массив того же типа, что и у исходного массива и с теми же элементами, кроме заменяемых элементов. В списке расположенных элементов не разрешается использовать описание «*else*».

```
// выражение let вычисляет значения:
// [1, 2, 0, 4, 5], [1, 60, 70, 20, 10], [1, 2, 3, 4, 5]
let A := [1, 2, 3, 4, 5]
in A[3 := 0], A[2..5 := 60, 70, 20, 10], A end let,
// выражение let вычисляет значения:
// A и array [1..2, -3..-2] of [1,.. := 0, 2; 2,.. := 3, 0]
let A := array [1..2, -3..-2] of [1,.. := 1, 2; 2,.. := 3, 4]
in A, A[1..2 dot -3..-2 := 0] end let
```

Для двух массивов определена бинарная операция конкатенации («||»), возвращающая одномерный массив, склеенный из двух указанных массивов, элементы которых располагаются в row-major порядке. Программа неправильна, если типы массивов неэквивалентны и не существует подходящей операции неявного преобразования типов элементов массивов[41].

```
// данное выражение истинно
let A1 := [10]; A2 := array [..1, ..2] of [:= 20, 30];
    A3 := [40]; A4 := array of integer []
in A1 || A2 || A3 || A4 end let = [10, 20, 30, 40]
```

Если тип элемента массива допускает префиксные и постфиксные[42] операции, то они допустимы и для этого массива, при этом порождается массив с нижней границей как у исходного и элементами, полученными после поэлементного применения операции над значениями исходного массива. Не допустимы постфиксные операции вызова функции более чем с одним результатом.

Если типы элементов двух массивов допускают инфиксные операции, то они допустимы и для этих массивов. При этом порождается массив с нижней границей, равной единице, и элементами, полученными при поэлементном применении операции без учета нижних границ, причем элементы меньшего по размеру массива дополняются ошибочными значениями. Определена операция неявного преобразования типа массива к типу потока, располагающая элемент массива в row-major порядке.

---

[40] Список расположенных элементов описывался в разделе 4.3.1.

[41] Сначала проверяется возможность неявного преобразования типа элемента второго массива к типу элемента первого массива, а потом возможность обратного преобразования.

[42] Кроме постфиксной операции квадратных скобок («[]»), так как она конфликтует с выражением выбора и замещения элементов массива.

Если тип элемента массива и значения допускают инфиксные операции, то они допустимы для этого значения и массива, порождая массив с нижней границей исходного массива и элементами, полученными при поэлементном применении операции для исходного массива и значения.

```
// выражение let вычисляет значения [false, true], [2, 4, 6] и
// array [1..2, 1..2] of [:= 8.0, 10.0, 12.0, 14.0]
let A1 := [1, 2, 3];
    A2 := array [5..6, -1..0] of [5,.. := 4, 5; 6,.. := 6, 7];
    A3 := [true, false]
in ~A3, A1 + A1, A2 * 2.0 end let
```

Для массива определяется функция одного аргумента «size», которая берёт на входе массив и возвращает количество элементов во всех его размерностях. Функция «size» определена для двух элементов и возвращает количество элементов его размерности, указанной вторым аргументом целого типа. Для массива определяется функция одного аргумента «transpose», которая берёт на входе массив и возвращает транспонированный массив.

Для массива определяется функция одного аргумента «liml», которая берёт на входе массив и возвращает нижнюю границу его первой размерности. Функция «liml» определена для двух элементов и возвращает нижнюю границу его размерности, указанной вторым аргументом целого типа. Для массива определяется функция одного аргумента «limh», которая берёт на входе массив и возвращает верхнюю границу его первой размерности. Функция «limh» определена для двух элементов и возвращает верхнюю границу его размерности, указанной вторым аргументом целого типа.

Для массива определены функции «floor», «trunc», «abs», «min» и «max», выполняющие соответствующие операции поэлементно.

4.4. Записи

Тип записи «*record* [объявление полей записи]» задает декартово произведение типов своих полей, к значениям которых имеется прямой доступ по их уникальному в пределах одного типа записи имени. Объявление полей записи содержит разделяемые точкой с запятой объявления полей с одинаковым типом «список имён полей : тип полей». Тип записи содержит ошибочное значение. Если запись ошибочна, то значение всех её полей тоже ошибочно. Типы записей эквивалентны, если они имеют одинаковое количество полей и типы соответствующих полей эквивалентны.

```
// рекурсивное определение записи, в отличие от рекурсивного
// определения объединения, не допустимо
```

```
type bad_stack := record [ value: real; rest: bad_stack]
// имя типа записи может быть использовано в качестве имени её поля
type Ex1 = record [Ex1: record [ Ex1: real ]]
// имя поля записи может быть использовано в другой записи
type Ex2 = record [Ex1 : Ex1]
// типы записей XYrec и YXrec не эквивалентны
type XYrec = record [ X: real; Y: integer ]
type YXrec = record [ X: integer; Y: real ]
// типы записей XYrec и ABrec эквивалентны
type ABrec = record [ A: real; B: integer ]
```

Запись конструируется как «*record* тип записи [определение полей записи]», «*record* [определение полей записи]» или «*record* тип записи [:= список выражений]». Определение полей записи содержит разделяемые точкой с запятой определения одного или нескольких полей «список имён полей := список выражений», указанных их именами. Для задания полей записи, являющейся полем записи, в качестве имени поля можно использовать несколько имён полей, разделённых точкой. Список выражений должен определять все указанные поля записи, а в выражении конструктора записи единственным образом должны быть определены все её поля. Значение выражений может неявно приводиться к типам полей, указанных типом записи.

```
// различные способы построения одной записи
record XYrec [X:= 2.0; Y: 1], record XYrec [Y: 1; X:= 2.0],
record XYrec [X, Y := 2.0, 1], record XYrec [Y, X := 1, 2.0],
record XYrec [:= 2.0, 1],
// построение записи с указанием вложенных полей
record Ex1 [Ex1.Ex1 := 1.0],
// построение записи record [a: real; b: integer] «на месте»
record [a:= 1.0; b := 1]
```

Значение поля записи можно получить как «запись . имя поля этой записи». Определена операция «замещения» полей записи «запись *replace* [определение полей записи]», которая создает новую запись такого же типа, но с новыми значениями полей, указанных их именами. Если запись является ошибочным значением, то порождается также ошибочное значение.

```
// получение значение поля записи
record XYrec [Y, X := 1, 2.0] . X, // выражение равно 2.0
// выражение равно record [ Ex1 := 1.0 ]
record Ex1 [Ex1.Ex1 := 1.0] . Ex1,
record Ex1 [Ex1.Ex1 := 1.0] . Ex1 . Ex1, // выражение равно 1.0
// выражение равно record XYrec [:= 1.0, 1]
record XYrec [:= 2.0, 1] replace [X := 1.0],
// выражение равно record Ex1 [Ex1.Ex1 := 2.0]
record Ex1 [Ex1.Ex1 := 1.0] replace [Ex1.Ex1 := 2.0]
```

4.5. Объединения

Тип объединения «*union* [объявление тегов объединения]» аналогичен типу записи, исключая то, что не более чем одно значение его тега может быть отлично от ошибочного значения. Объявление тегов объединения содержит разделяемые точкой с запятой объявления тегов с одинаковым типом «список имён тегов : тип тегов». При объявлении тегов объединения можно опускать их типы (вместе с двоеточием), если они равны null. Тип объединения содержит ошибочное значение. Если объединение ошибочно, то значение всех его тегов тоже ошибочно. Типы объединений эквивалентны, если они имеют одинаковое количество тегов и типы соответствующих тегов эквивалентны.

```
// примеры определения объединений
type UnEx1 = union [ T1: real;    T2: integer ]
type UnEx2 = union [ T1: integer; T2: UnEx1 ]
type StNode := union [ Empty; Element: record [Value: real; Next: StNode] ]
type UnType := union [ red, green, blue, black, white ]
```

Объединение конструируется как «*union* тип объединения [имя тега этого объединения := значение тега]», где значение тега вместе с со знаком присваивания можно опускать, если тип тега равен типу null. Значение тега может неявно приводиться к типу тега объединения. Выражение «объединение *is tag* имя тега этого объединения» истинно, если имя тега равно имени тега данного объединения; ложно, если не равно и ошибочно, если само объединение ошибочно. Конструкция «объединение . имя тега этого объединения» равна значению, заданному указанным именем тега объединения.

```
// примеры конструирования объединений
union UnEx2 [ T1 := 1 ],
union UnEx2 [ T2 := union UnEx1 [ T1 := 2.0 ] ],
union UnType [ red ],
// примеры проверки тегов объединений
union UnEx1 [ T2 := 1] is tag T2, // выражение равняется true
union UnEx1 [ T2 := 1] is tag T1, // выражение равняется false
// примеры получения значений тегов объединений
union UnEx1 [ T2:= 1] . T2, // выражение равняется 1
union UnEx1 [ T2:= 1] . T1 // выражение равняется error[real]
```

4.6. Функции

Тип функции задается как «*function* [типы аргументов *returns* типы результатов]», содержит все процедуры с указанными типами аргументов и результатов и ошибочное значение. Значение типа функции можно сконструировать с помощью:

- имени объявленной необобщенной функции «имя функции»[43], если имя функции и имя модуля не перекрыто локальным именем и функция задаётся однозначно[44];
- имени функции «*function* имя функции [..]», позволяющего указать однозначно заданную необобщенную функцию, даже если она и имя её модуля перекрыто локальным именем;
- имени обобщенной функции «имя функции . [типы свободных параметров]» только со свободными параметрами, которая задана однозначно (даже если она и имя её модуля перекрыто локальным именем);
- конструкции «*function* имя функции [список типов формальных параметров]» для указания конкретной необобщенной функции однозначной по возвращаемым значениям или, в случае её отсутствия, максимально простой[45] обобщенной функции без свободных параметров и однозначной по возвращаемым значениям, в которой значения параметров восстанавливаются путём прохода по типам формальных параметров слева направо;
- конструкции «*function* имя функции [список типов формальных параметров *returns* список типов возвращаемых значений]» для указания конкретной необобщенной функции или, в случае её отсутствия, максимально простой обобщенной функции;
- конструкции «*function* имя функции [список типов формальных параметров инородной функции]» для указания инородной процедуры (инородная операция указывается её именем функции)[46], где типы формальных параметров списка формальных параметров функции могут предваряться ключевыми словами «*raw*», «*in*», «*out*» и «*in out*»;

---

[43] Имя функции может всегда указываться с помощью имени модуля «имя модуля . имя функции».

[44] Операция вызова функции позволяет задавать обобщенную функцию с помощью неоднозначного имени.

[45] Максимальной простой обобщенной функцией среди всех других обобщенных функций с тем же именем, которые можно применить при определённом обозначении параметров, является та, которая содержит минимальное число несвободных параметров.

[46] Инородной функции соответствует тип функции, полученный после добавления возвращаемых значений для формальных параметров с ключевыми словами «*in out*» и «*out*», удаления формальных параметров только с ключевым словом «*out*» и удаления всех ключевых слов «*raw*», «*in*», «*out*» и «*in out*».

- конструкции «*operation* знак операции [тип формального параметра *returns* тип возвращаемого значения]» для указания операций[47] со знаками «:» и «ε» и конструкции «*operation* знак операции [список типов формальных параметров]» для указания операций с другими знаками;
- конструкции «*function* имя функции . [типы свободных параметров] [список типов формальных параметров]» для указания максимально простой обобщённой функции со свободными параметрами, однозначной по возвращаемым значениям;
- конструкции «*function* имя функции *of* [имена и типы параметров контракта] [список типов формальных параметров]» для указания конкретной обобщённой функции, однозначной по возвращаемым значениям, где имена и типы параметров контракта содержат перечисляемые через запятую структуры вида «имя = тип», которые можно сокращать до «тип» для свободных параметров, а список типов формальных параметров зависит от имён указанных параметров контракта так же, как и у обобщенной функции, которую нужно указать;
- конструкции «*function* имя функции *of* [имена и типы параметров контракта] [список типов формальных параметров *returns* список типов возвращаемых значений]» для указания конкретной обобщённой функции;
- конструкции «*operation* знак операции *of* [имена и типы параметров контракта] [тип формального параметра *returns* тип возвращаемого значения]» для указания конкретных обобщённых операций со знаками «:» и «ε» и конструкции «*operation* знак операции *of* [имена и типы параметров контракта] (список типов формальных параметров)» для импорта обобщённых операций с другими знаками;
- конструкции «*function* имя функции (список формальных параметров *returns* типы результатов) список выражений *end function*» для определения λ-функции, в которой имя функции необязательно и используется в списке выражений функции только для задания рекурсивной зависимости (в списке выражений функции разрешается использовать имена значений, определённых вне тела функции).

[47] Обобщенная операция может быть указана таким образом, если не существует подходящей необобщенной операции и существует максимально подходящая обобщенная функция.

```
type tfoo1 = function [integer returns integer]
type tfoo2 = function [real returns real]
type tfoo3 = function [integer returns integer, integer]
type tfoo4 = function [integer, integer returns integer, integer]
type tfoo5 = function [returns integer]

function foo1 (i: integer returns integer) i end function
function foo2 (i: integer returns integer) i end function
function foo2 (r: real returns real) r end function
function foo3 (i: integer returns integer) i end function
function foo3 (i: integer returns integer, integer) i, i end function
function foo4 of any[T] (t: T returns T) t end function
function foo5 of any[T] (t: T returns T) t end function
function foo5 of any[T] (t1: T, t2: T returns T, T) t1, t2 end function
function foo6 of any[T] (t: T returns T) t end function
function foo6 of any[T] (t: T returns T, T) t, t end function
function foo7 of any[T] (returns T) error[T] end function
function foo8 of any[T] (returns T) error[T] end function
function foo8 of any[T] (integer returns T) error[T] end function

function test1 (foo1: integer returns integer, tfoo1, tfoo2, tfoo3,
                                       tfoo1, tfoo4, tfoo1, tfoo5,
                                       tfoo5, tfoo1, tfoo1, tfoo1)
  foo1,
  function foo1 [..],
  function foo2 [real],
  function foo3 [integer returns integer, integer],
  function foo4 of [T=integer] [T],
  function foo5 of [T=integer] [T, T],
  function foo6 of [T=integer] [T returns T],
  function foo7 . [integer] [],
  foo7 . [integer],
  function foo8 . [integer] [integer],
  function (i: integer returns integer) i + foo1 end function,
  function fact (n: integer returns integer)
    if n = 1 then 1 else fact(n-1)*n end if
  end function
end function
```

Тип функции имеет операцию её вызова «функция (значения аргументов)», возвращающую результаты функции, вычисленные после подстановки значений аргументов на место имён формальных параметров. Если значение функции ошибочно, то результаты операции её вызова будут ошибочны.

```
function test2 (returns integer, real, integer)
  foo2(1), foo2(1.0), foo1(1.0)
end function
```

Операция вызова функции автоматически разрешает неоднозначность функции, заданной её именем, на основании типов ее аргументов (если функция однозначна по возвращаемым значениям). Если не существует функции с формой, состоящей из типов, эквивалентных типам аргументов, то выбирается функция с минимальным количеством[48] неявных преобразований, которое необходимо осуществить для преобразования типов значений аргументов в типы формальных параметров.

Если не существует подходящей необобщенной функции, и функция задана своим именем, то операция вызова функции может указать максимально подходящую обобщенную функцию без свободных параметров. Параметры обобщенных функций восстанавливаются путём прохода по типам формальных параметров слева направо с попытками применения неявных преобразований типов значений аргументов в типы формальных параметров. Если отсутствует обобщенная функция, требующая минимального количества неявных преобразований, то среди обобщенных функций с наименьшем количеством неявных преобразований выбирается максимально простая обобщенная функция.

```
function test3 (returns real)
  foo5(1.0, 1) // вызывается функция «function foo5 of [T=real] [T, T]»
end function
```

Однозначно заданную обобщенную функцию со свободными параметрами можно вызвать следующим образом: «имя функции . [типы свободных параметров] (значения аргументов)». Типы свободных параметров указываются в порядке их перечисления в контракте объявления обобщенной функции[49]. В остальном вызов обобщенной функции со свободными параметрами аналогичен вызову обобщенной функции без свободных параметров.

```
function test4 (returns real)
  // вызывается функция «function foo8 of [T=real] [integer returns T]»
  foo8.[real](1)
end function
```

[48] С учётом количества неявных преобразований, полученных при применении свойства дистрибутивности неявного преобразования.

[49] Порядок следования типов свободных параметров гарантированно сохраняется при переобъявлении обобщенной функции.

Если аргумент функции пропущен, то результатом операции вызова функции, называемой теперь «сужением» области определения функции, будет функция от пропущенных аргументов в порядке их следования и с результатами исходной функции. Если значение функции ошибочно, то операция её сужения возвратит ошибочное значение функции. Операция сужения функции также разрешает неоднозначность функции на основании типов её аргументов, указываемых как «: тип» для пропускаемых аргументов.

```
function test5 (returns function [integer returns integer, integer],
                function [real returns real, real])
  foo5(1,), foo5(:real,1)
end function
```

4.7. Пользовательские типы

Пользовательские типы отличаются от всех прочих тем, что их необходимо определять от некоторого базового типа, который тоже в свою очередь может быть пользовательским типом. Пользовательский тип **A** основывается на другом пользовательском типе **B**, если тип **B** является базовым типом типа **A**, или тип **B** является базовым типом пользовательского типа, на котором основывается тип **A**. Встроенный тип **B** лежит в основе пользовательского типа **A**, если он является базовым типом типа **A**, или он является базовым пользовательского типа, на котором основывается тип **A**. Пользовательский тип является подмножеством значений своего базового типа.

```
// пример пользовательского типа, поддерживающего сумму двух чисел
type sum_pair := record [ a, b, sum: integer ]
```

Для пользовательского типа не определяется никаких встроенных операций[50], кроме как операций явного преобразования базового типа к пользовательскому типу и операции явного преобразования пользовательского типа к пользовательскому типу, на котором он основывается, и к встроенному типу, лежащему в его основе. Непереопределяемые операции явного преобразования пользовательского типа к пользовательскому типу, на котором он основывается, и к встроенному типу, лежащему в его основе, возвращают значение, из которого данное значение пользовательского типа было сконструировано. Операцию явного преобразования базового типа к

[50] Операции базового типа пользовательским типом не наследуются, и для их использования необходимо выполнить преобразование к базовому типу.

пользовательскому типу можно переопределить[51], и в её определении происходит неявное преобразование значения базового типа в значение пользовательского типа.

```
// конструктор, гарантирующий правильное состояние поля sum типа sum_pair
operation : (base: record [ a, b, sum: integer ] returns sum_pair)
  base replace [sum := base.a + base.b]
end operation
```

Ошибочное значение пользовательского типа конструируется путём применения операций явных преобразований типов к ошибочному значению типа, лежащего в основе пользовательского типа.

4.8. Инородные типы

Инородной тип задаётся строкой «“строковое представление инородного типа”», где строковое представление инородного типа должно задаваться на языке Си++ для использования в инородных функциях на языке Си++ и на подмножестве языка Си++, соответствующего языку Си, для инородных функций на языке Си и Фортран[52]. Инородные типы эквивалентны, только если совпадает их строковое представление, что является более строгим критерием эквивалентности, чем это необходимо, и в обязанности программиста входит поддержание его правильности.

```
// типы int1 и int2 считаются разными в языке Sisal,
// но в языке Си это не так
type int1 = "long"
type int2 = "long int"
```

Никаких операций для инородного типа изначально не определено. Значения инородных типов конструируются только в инородных процедурах. Если для инородного типа **T** определена операция «*operation* (**T** *returns* **T**)», то она используется для создания копии значения инородного типа **T**. Если операция копирования запрещена («*no operation* (**T** *returns* **T**)»), то копирование значения инородного типа тоже запрещено. Если операция копирования не определена и не запрещена, то используется побитовое копирование значения инородного типа.

---

[51] Переопределение операции явного преобразования базового типа к пользовательскому типу позволяет задать ограничения на содержимое (правильность) базового типа, которое невозможно обойти при создании пользовательского типа.

[52] Вызов функций и процедур языка Фортран возможен только для процедур, использующих средства взаимодействия с языком Си, появившихся в языке Фортран-2003.

Если для инородного типа **T** определена операция «*operation* (**T** *returns* null)», то она используется для освобождения копии значения инородного типа **T**, иначе никаких специальных действий для освобождения копии инородного типа не выполняется. Ошибочное значение инородного типа **T** соответствует его неопределенному значению, если не определена операция «*operation* (null *returns* **T**)», которая возвращает значение инородного типа **T**, соответствующее ошибочному значению инородного типа.

```
// строки в динамической памяти, оканчивающиеся нулевым символом
type psz := "char*"
operation (array of character returns psz)
operation (psz returns array of character)
operation (psz returns psz)  // копирование строк возможно
operation (psz returns null)
operation (null returns psz) // error[psz] это нулевой указатель
```

5. ВЫРАЖЕНИЯ

Выражения языка Sisal могут быть n-арными. Любое унарное (n=1) выражение языка Sisal рассматривается как арифметическое выражение. Список выражений задаётся перечисленными через запятую выражениями. Размерность списка выражений равна сумме размерностей выражений, в него входящих.

```
// размерность данного списка выражений равна трём
if a < b then a, b else b, a end if, 1
```

Перед каждым выражением могут располагаться прагмы[53] «assert = булевское условие», которые могут проверяться на истинность компилятором после вычисления выражения и использоваться при оптимизирующих преобразованиях программы. Результат унарного выражения в булевском выражении обозначается через одиночный символ подчеркивания «_». Арности n-арного (n≥1) выражения обозначаются как «_[1]», …, «_[n]».

Прагмы «pre_assert = булевское условие» и «post_assert = булевское условие» могут располагаться перед ключевым словом «*returns*» в объявлениях процедур и накладывать условия на возвращаемые значения и указанные имена формальных параметров, проверяемые до (pre_assert) или после (post_assert) вызова процедуры.

[53] Подробнее о прагмах можно прочитать в разделе 7.3.

```
// факториал числа n
forward function fact (n: integer
/*$ assert = n >= 1*/ /*$ assert = _ >= n*/
  returns integer)

function fact (n: integer returns integer)
  if n = 1 then 1 else /*$ assert = _ > 0*/ fact(n-1)*n end if
end function
```

5.1. Арифметическое выражение

Арифметическое выражение содержит операнды и операции. Операндами являются унарные выражения. Операции могут быть постфиксными, префиксными и инфиксными.

Постфиксные операции имеют вид «операнд операция». Цепочка постфиксных операций вычисляется слева направо до начала вычисления префиксных операций. К постфиксным операциям относятся операции вызова и «сужения» функции, выбора и замены элементов массива, выбора элементов потока, доступа к полю записи или объединения, замены элементов записи, проверка тега объединения и операция явного приведения типов.

Префиксные (унарные) операции имеют вид «знак операции операнд». Цепочка префиксных операций вычисляется справа налево до начала вычисления инфиксных операций. К префиксным операциям относятся операции смены знака («-»), идентичности («+») и логического отрицания («!»).

Инфиксные (бинарные) операции имеют вид «операнд знак операнд». Среди нескольких инфиксных операций раньше выполняются операции на более глубоком уровне вложенности арифметических скобок. Среди инфиксных операций одного уровня вложенности сначала выполняются операции с большим приоритетом, указанным в таблице 3 . Лево-связываемые операции одного приоритета выполняются слева направо, а право-связываемые операции — справа налево. Цепочка операций сравнения объединяется операциями конкатенации, например «A < B <= C» рассматривается как «A < B & B <= C».

```
(1+(3+5)*10)/3 ** 2 // в результате получаем число 9
```

Таблица 3

Свойства инфиксных операций

Приоритет	1	2	3	4	5	6	7	8	9
Знак	||	|	^	&	= !=	< > <= >=	- +	* / %	**
Связывание	«Левое»								«Правое»

Если для операндов инфиксной операции не объявлена операция для типов, эквивалентных типам операндов, то делаются попытки неявных преобразований и применений операции над получившимися типами. Сначала делается попытка неявного преобразования второго операнда к типу первого операнда, а потом попытка неявного преобразования первого операнда к типу второго операнда.

Операция неявного преобразования типов дистрибутивна. Если определена операция неявного преобразования типа **A** в тип **B** и операция неявного преобразования из типа **B** в тип **C**, то определена операция неявного преобразования типа **A** в тип **C** через тип **B**. Действует правило применения минимального неявного преобразования, которое вызовет непосредственно преобразование из типа **A** в тип **C**, если оно определено.

5.2. Выражение «let»

Выражение «*let*» определяет новую область и множество ее имен, используя их для вычисления списка выражений своих результатов: «*let* определения имен *in* список выражений результатов *end let*». Определения имен содержат разделенные точкой с запятой определения, содержащие левую и правую части, разделенные символами знака равенства с двоеточием. Левая часть — это разделенные запятыми определяемые имена, после каждого из которых может явно указываться его «: тип». Правая часть состоит из списка выражений, сумма размерностей которых равна числу имен левой части определения. Выражения правой части определения не могут зависеть от имен его левой части. Область действия определённых имён состоит из правых частей последующих определений и списка выражений результатов.

```
// выражение равно 3.0 * G * 3.0, 3.0
let X := 3.0; A := X * G in A * X, X end let
// данное выражение равно 4
let A := 3 in let A := A + 1 in A end let end let
```

5.3. Выражение «if»

Выражение «*if*» выглядит как «*if* булевское условие *then* список выражений результата ветви *elseif* ветвь *else* *end if*», где каждая из необязательных ветвей «*elseif*» задаётся как «*elseif* булевское условие *then* список выражений результата», а необязательная ветвь «*else*» задаётся как «*else* список выражений результата».

У всех списков выражений результата размерности должны быть равны. Типы возвращаемых значений выражения «*if*» определяются типами размерностей в первом списке выражений результата (после ключевого «*then*»). Типы размерностей в других списках выражений результата должны быть эквивалентны типам соответствующих результатов выражения «*if*» или неявно к ним приводиться.

Выражения булевских условий вычисляются последовательно, пока не получится истинное значение. Список выражений результата, идущий за первым истинным булевским условием, определяет результаты выражения «*if*». Если все булевские условия ложны, то ветвь «*else*» определяет результат конструкции. Если ветвь «*else*» отсутствует или встретилось ошибочное значение булевского условия, то выражение «*if*» возвращает ошибочные значения.

Ниже приведено простое выражение «*if*», описывающее модуль числа x:

```
if x < 0 then -x else x end if
```

Далее приведено более сложное выражение «*if*», вычисляющее корни квадратного уравнения в зависимости от знака дискриминанта:

```
let d := b**2 - 4*a*c
in if    d > 0 then (-b+d**0.5)/2*a, (-b-d**0.5)/2*a
  elseif d = 0 then          -b/2*a,          -b/2*a
  else                  error[real],     error[real]
  end if
end let
```

5.4. Выражение «case»

Выражение «*case*» выглядит как «*case* управляющее выражение условные ветви ветвь *else end case*», где должна присутствовать хотя бы одна условная ветвь вида «*of* список значений тестов *then* список выражений результата», а необязательная ветвь «*else*» задаётся как «*else* список выражений результата». Управляющее выражение должно быть унарным.

У всех списков выражений результата размерности должны быть равны. Типы возвращаемых значений выражения «*case*» определяются типами размерностей в первом списке выражений результата. Типы размерностей в других списках выражений результата должны быть эквивалентны типам соответствующих результатов выражения «*case*» или неявно к ним приводиться.

Значение теста может быть унарным выражением и, если тип управляющего выражения имеет операции со знаками «=» и «<», дуплетом «нижняя граница .. верхняя граница», в котором опущенные унарные выраже-

ния нижней и верхней границы по умолчанию равняются минус и плюс бесконечности. Если значение теста является унарным выражением, то тестом является сравнение его значения со значением управляющего выражения. Если значением теста является дуплет, то тестом является булевское выражение «нижняя граница <= управляющее выражение <= верхняя граница».

Значение управляющего выражения проверяется тестом каждой условной ветви в порядке их следования до первого совпадения. Список выражений результата, идущий за первым истинным тестом, определяет результаты выражения «*case*». Если все тесты ложны, то ветвь «*else*» определяет результат конструкции. Если ветвь «*else*» отсутствует или значение теста ошибочно, то выражение «*case*» возвращает ошибочные значения.

```
case die_1 + die_2
  of 2..3,    12 then        "lose"
  of    7,    11 then         "win"
  of 4..6, 8..10 then "no decision"
  else error[array of character]
end case
```

Если известно, что истинным может быть только один тест, то для выражения «*case*» можно указать прагму «parallel» или прагму «parallel = булевское условие», если тест может быть истинным при определённом булевском условии.

Для выбора условий по тегам объединения существует выражение «*case tag*», выглядящее как «*case tag* управляющее выражение типа объединение условные ветви выражения *case tag* ветвь *else end case*». Значениями тестов выражения «*case tag*» являются имена объединения, указанного управляющим выражением. Значения тестов не должны повторяться.

```
type NodeType := union[tail; link: NodeType; data: integer]
… // значение node принадлежит типу NodeType
case tag node : union[tail; link: NodeType; data: integer]
  of link then Traverse(node.link)
  of data then node.data
  of tail then 0
end case
```

5.5. Циклические выражения

Форма циклического выражения такова: «заголовок цикла тело цикла эпилог цикла». Заголовок цикла может быть пустым или задаваться следующим образом: «*for* генератор диапазона ; определения имён начальных значений», «*for* генератор диапазона», «*for* определения имён начальных

значений», где генератор диапазона должен завершаться точкой с запятой, если после него идёт непустое тело цикла[54]. Тело цикла может быть пустым или задаваться следующим образом: «тест», «тест *do* определения имён циклических значений», «*do* определения имён циклических значений тест» или «*do* определения имён циклических значений». Если заголовок цикла не содержит генератор диапазона, то тело цикла должно содержать тест. Тест имеет вид «*while* булевское условие» или «*until* булевское условие». Эпилог цикла имеет вид «*returns* список редукций *end* первое ключевое слово циклического выражения[55]», где элементы списка редукций разделяются запятыми.

Например, в данном примере итеративно вычисляется число π:

```
for Approx := 1.0; Sign := 1.0; Denom := 1.0; i := 1
while i <= Cycles do
  Sign := - old Sign; Denom := old Denom + 2.0;
  Approx := old Approx + Sign / Denom;
  i := old i + 1
returns value of Approx * 4.0
end for
```

В следующем примере также итеративно вычисляется число π (с тем же результатом что и в предыдущем примере при чётном значении Cycles):

```
for i in 1..Cycles/2; do val := 1.0 / (4*i-3):real — 1.0 / (4*i-1):real
returns sum of val end for * 4.0
```

Циклическое выражение параллельно, если оно не содержит теста, его генератор диапазона не перечисляет потоки, оно не содержит редукций «*stream*», все функции начальных значений пользовательских редукций помечены прагмой «identity», все собирающие функции пользовательских редукций помечены прагмой «associative», оно не содержит «*old*» имён, определения имён циклических значений в правых частях не содержат имён, которые потом определяются в левой части. Циклическое выражение асинхронно параллельно, если все собирающие функции пользовательских редукций дополнительно помечены прагмой «commutative».

5.5.1. Заголовок и тело цикла

Циклическое выражение управляется тестом или генератором диапазона или тем и другим одновременно. Циклическое выражение, управляемое

[54] Для устранения неоднозначности разбора последнего триплета в генераторе диапазона.

[55] Первым ключевым словом циклического выражения могут быть ключевые слова «*for*», «*while*», «until» и «*do*».

тестом, выполняется пока «*when*» булевское условие истинно или «*until*» булевское условие ложно. Условие проверяется до или после тела цикла. Циклическое выражение, управляемое диапазоном, выполняется до тех пор, пока все элементы диапазона не будут перебраны. Если циклическое выражение управляется тестом и диапазоном одновременно, то оно выполняется до тех пор, пока позволяет тест и диапазон.

Генератор диапазона задаётся понятием «положение», введенным в разделе 4.3.1 для выражений конструирования массивов со следующими изменениями. Нет ограничений на количество элементов списка, задаваемого количеством размерностей массива. Элементы списка положения разделяются не запятой, а ключевым словом «*cross*». Элементами списка могут быть только обязательно именованные триплеты, потоки и массивы (не обязательно целого типа). Выражения не могут быть элементами списка. Неуказанные нижняя и верхняя границы триплетов генератора диапазона равны единице и бесконечности (положительной для положительного шага и отрицательной для отрицательного шага).

```
i in 1..3         // 1, 2, 3
j in ..3          // 1, 2, 3
k in 3..100..2    // 3, 5, …, 99
l in 100..98..-1 // 100, 99, 98
m in 1..          // 1, 2, 3, …
n in ..           // 1, 2, 3, …

// следующее выражение суммирует числа от 1 до N
for i in 1..N returns sum of i end for
// следующее выражение суммирует значения потока,
// удовлетворяющие контракту add1 из раздела 5.5.4
for x in S returns sum of x end for
// следующее выражение равно 24 (выполняется две итерации [1,3] и [2, 4])
for i in 1..2 dot j in 3..4 returns product of i+j end for
// следующее выражение равно 600 (итерации [1,3], [1, 4], [2, 3] и [2, 4])
for i in 1..2 cross j in 3..4 returns product of i+j end for
// следующее выражение равно 14400
for i in 1..2 cross j in i..4 returns product of i+j end for
```

Для массивов генератора диапазона допускается указание перечисляемых размерностей путём добавления суффикса «*at* [список имён индексов перечисляемых размерностей]». Список имён индексов перечисляемых размерностей должен иметь число элементов, равное размерности массива, и содержать, по крайней мере, одно имя индекса. Не перечисляемые размерности обозначаются двоеточием «..». Имя массива задаёт значение, полученное выражением выбора элементов массива с положением, равным списку имён индексов перечисляемых размерностей. Если суффикс «*at*»

отсутствует, то перечисляются все размерности, и имя массива имеет тип элемента массива. Массив генератора диапазона задаёт количество размерностей формы положения, равное числу своих перечисляемых размерностей.

```
// для трёхмерного массива «A» значение «x» является двухмерным массивом
// значение «i» пробегает все индексы первой размерности массива «A»
for x in A at i,..,..; do … x[..,..] … returns … end for
// в следующем примере перебираются все элементы массива «A», что
// эквивалентно заголовку цикла «for x in A at [i,j,k]»
for x in A; do … x … returns … end for
```

Определения имён начальных значений семантически полностью эквивалентны их заданию в окружающем выражении «*let*». Имена начальных значений и имена значений, определённых извне цикла, называются константами цикла. Определения имён циклических значений на каждой итерации цикла переопределяет значения указанных имён, так что на следующей итерации эти имена, использованные раньше в повторяемой части циклического выражения, будут иметь новые значения (определяемые имена должны иметь тип, эквивалентный или неявно приводимый к типу константы цикла с этим именем). После определения (текстуально, начиная с правой части этого определения до окончания циклического выражения) циклического имени можно обращаться к его значению на предыдущей итерации через «*old* имя», если существует константа цикла с этим именем.

```
// следующее выражение равно 91
for i := 1 while i < 5 do k := i; i := old i + 2; j := k + i
returns product of i+j end for
// следующее выражение цикла семантически эквивалентно предыдущему
// выражение «let» возвращает значения 91, 1
let i := 1 in while i < 5 do k := i; i := old i + 2; j := k + i
              returns product of i+j end while,
              I
end let
```

5.5.2. Эпилог цикла

Итерации цикла определяют редуцируемую последовательность значений для каждого имени, задаваемого генератором диапазона и телом цикла. Редукции формируют из редуцируемых последовательностей одно или несколько возвращаемых значений циклического выражения. Редукция имеет вид «*stream of* выражение фильтр», «*array of* выражение фильтр», «*array* форма свободного вида *of* выражение», «*array* форма фиксированного вида *of* выражение *at* список выражений положения» или «пользовательская редукция фильтр». Фильтр имеет вид «*when* булевское условие» или «*unless*

булевское условие» и включает циклические значение в редуцируемую последовательность только в случае, если «*when*» булевское условие истинно или «*unless*» булевское условие ложно.

Редукция «*stream*» формирует поток из своей редуцируемой последовательности. Редукция «*array*» без формы формирует одномерный массив с нижней границей равной единице из своей редуцируемой последовательности. Редукция «*array*» с формой свободного вида может использоваться только при отсутствии теста цикла и должна задавать форму с размерностью, равной размерности формы генератора диапазона. Редукция «*array*» с формой свободного вида конструирует массив с формой генератора диапазона, элементы которого задаются редуцируемой последовательностью и укладываются в конструируемый массив согласно форме генератора диапазона.

Границы формы фиксированного вида редукция «*array*» должны задаваться константами цикла. По-умолчанию, необязательная нижняя граница размерностей предполагается равной единице. Список выражений положения должен содержать число выражений, равное размерности конструируемого массива. Выражения должны быть целого типа и для каждой итерации цикла задавать различные элементы массива, иначе возвращается ошибочное значение массива. Все незаданные элементы массива (включая элементы с ошибочным или выходящим за границы массива положением) равняются ошибочным значениям.

```
// следующее выражение возвращает значение stream [2, 3, 4]
for i in 1..2 cross j in i..2 returns stream of i+j end for
// следующее выражение возвращает значение array of [2, 3, 4]
for i in 1..2 cross j in i..2 returns array of i+j end for
// следующее выражение возвращает значение array of [2, 3, 4]
for i in 1..2 cross j in i..2
returns array [1..3] of i+j at (i-1)*2 + j end for
// следующее выражение строит две одинаковые матрицы
let A := array [1..2, 0..1] of [1,.. := 1, 2; 2,.. := 3, 4];
    B := for i in 1..4
         returns array [1..2, 0..1] of i at i / 2 + 1, 1 - i % 2 end for
in A, B end let
// следующее выражение возвращает значение [3, 4, 6, 8]
for i in 1..2 cross j in 3..4 returns array of i*j end for
// выражение определяет массив с формой и границами массива «A»
for x in A at i, j; do k := … returns array [..,..] of g(x, k) end for
// следующее выражение возвращает значение array [4..7] of [40, 30, 20, 10]
for i in 1..4 returns array [4..7] of i * 10 at 4+i-1 end for
```

Пользовательские редукции рассматриваются в разделе 5.5.3. Предопределяются редукция «value», возвращающая последнее значение редуцируемой последовательности, редукция «sum», возвращающая сумму значе-

ний редуцируемой последовательности, редукция «product», возвращающая произведение значений редуцируемой последовательности, редукция «greatest», возвращающая наибольшее значение редуцируемой последовательности, редукция «least», возвращающая наименьшее значение редуцируемой последовательности. Также предопределяется редукция «catenate», возвращающая поток или массив, склеенный из потоков или массивов редуцируемой последовательности.

Если редуцируемая последовательность значений для редукции пуста (генератор цикла задаёт пустой диапазон значений, тест перед циклом был не удовлетворён или фильтр не пропустил ни одного значения), то редукция возвращает значения по умолчанию. Редукция «*stream*» возвращает пустой поток. Редукция «*array*» возвращает пустой массив для массивов, у которых не задана форма фиксированного вида, а для массивов с формой фиксированного вида возвращается ошибочное значение. Для пользовательских редукций по умолчанию возвращается начальное значение.

```
// данное выражение возвращает массив нечетных целых чисел
for i in 1..N returns array of i when i % 2 != 0 end for
```

5.5.3. Пользовательские редукции

Редукцией является объединение функций специального вида, участвующих в формировании результатов циклических выражений. Конструкция вызова редукции в предложении возврата циклического выражения выглядит следующим образом: «имя редукции (список значений начальных параметров редукции) *of* (список значений циклических параметров редукции)» или «имя редукции *of* (список значений циклических параметров редукции)», если у редукции нет начальных параметров. Если указан один циклический параметр редукции, то скобки вокруг него можно опускать. Значения начальных параметров редукции должны задаваться константами цикла. Имя редукции может обозначать значение типа запись с именами полей «ini», «rep», «join» и «ret», которые должны содержать значения функций, назначение которых объяснено ниже. Вместо имени редукции можно использовать конструктор указанного типа записи.

Если имя редукции (которое может предваряться именем модуля «имя модуля .») обозначает имя функции **A** (а не значения), то в качестве функции («ini»), формирующей начальное редукционное значение типа **T**, берётся функция с именем **A** и формой ближе всего[56] к типам значений начальных параметров редукции. Данная функция должна возвращать одно

[56] Алгоритм определения наиболее подходящей формы функции находится в разделе 4.6.

значение типа **T**. В качестве функции, формирующей начальное редукционное значение, может использоваться (если не подходит ни одна другая функция) обобщенная функция со свободными параметрами, число которых равно количеству циклических параметров редукции. Свободные параметры обобщенной функции задают типы соответствующих циклических параметров редукции. Функция «ini» может быть помечена прагмой «identity», если возвращаемое значение является единичным значением типа **T** относительно операции «join», описываемой ниже: «join(a, ini()) = join(ini(), a) = a».

В качестве функции («rep»), перевычисляющей редукционное значение, берется функция с именем **A** с формой, которая содержит тип **T** в качестве первого параметра и другими типами ближе всего к типам значений циклических параметров редукции. Данная функция также должна возвращать одно значение типа **T**.

Если объявлена собирающая функция («join») с именем **A**, которая берёт на входе два аргумента типа **T** и возвращает значение типа **T** (эта функция может совпадать с функцией «rep»), то она используется для сбора различных значений редукции, вычисленных параллельно. Если начальное редукционное значение было построено функцией, не помеченной прагмой «identity», то определять отдельную функцию «join» смысла нет. Объявление функции «join» можно помечать прагмой «associative», если функция ассоциативна: «join(join(a, b), c) = join(a, join(b, c))». Объявление функции «join» можно помечать прагмой «commitative», если функция коммутативна: «join(a, b) = join(b, a))». Если функция «join» ассоциативна, то она будет использована для параллельного вычисления значений редукции, тем самым определять отдельную не ассоциативную функцию «join» смысла нет. Если функция «join» ассоциативна и коммутативна, то она будет использована для (более эффективного) асинхронного параллельного вычисления значений редукции. Прагмы «identity», «associative» и «commutative» собираются вместе со всех объявлений одной функции.

Результирующие значения редукции определяются возвращающими значениями функции («ret») с именем **A** и формой формальных параметров, состоящей из одного типа **T**.

5.5.4. Предопределённые редукции

В интерфейсе модуля «std» определяются следующие функции, задающие предопределённые редукции:

```
// редукция, возвращающая последнее из редуцируемых значений
// или ошибочное значение по умолчанию
type value_red[T] := T
contract any[T] end contract
function value of any[T] (returns value_red[T])
function value of any[T] (value_red[T], T returns value_red[T])
function value of any[T] (value_red[T] returns T)

// редукция, возвращающая сумму редуцируемых значений
// или нулевое значение по умолчанию
type sum_red[T] := T
contract add1[T]
  operation + (T, T returns T)
  function zero (returns T)
end contract
//$ identity
function sum of add1[T] (returns sum_red[T])
function sum of add1[T] (sum_red[T], T returns sum_red[T])
/*$ commutative*/ /*$ associative*/
function sum of add1[T] (sum_red[T], sum_red[T] returns sum_red[T])
function sum of add1[T] (sum_red[T] returns T)
function zero (returns integer)
function zero (returns real)

// редукция, возвращающая перемноженные редуцируемые значения
// или единичное значение по умолчанию
type mul_red[T] := T
contract mul1[T]
  operation * (T, T returns T)
  function one (returns T)
end contract
//$ identity
function product of mul1[T] (returns mul_red[T])
function product of mul1[T] (mul_red[T], T returns mul_red[T])
/*$ commutative*/ /*$ associative*/
function product of mul1[T] (mul_red[T], mul_red[T] returns mul_red[T])
function product of mul1[T] (mul_red[T] returns T)
function one (returns integer)
function one (returns real)

// редукция, возвращающая наименьшее из редуцируемых значений
// или максимальное  значение по умолчанию
type min_red[T] := T
contract cmp1[T]
  operation < (T, T returns T)
  function min (returns T)
  function max (returns T)
end contract
//$ identity
function least of cmp1[T] (returns min_red[T])
function least of cmp1[T] (min_red[T], T returns min_red[T])
```

```
/*$ commutative*/ /*$ associative*/
function least of cmp1[T] (min_red[T], min_red[T] returns min_red[T])
function least of cmp1[T] (min_red[T] returns T)
function min (returns integer)
function min (returns real)
function max (returns integer)
function max (returns real)

// редукция, возвращающая наибольшее из редуцируемых значений
// или минимальное значение по умолчанию
type max_red[T] := T
//$ identity
function greatest of cmp1[T] (returns max_red[T])
function greatest of cmp1[T] (max_red[T], T returns max_red[T])
/*$ commutative*/ /*$ associative*/
function greatest of cmp1[T] (max_red[T], max_red[T] returns max_red[T])
function greatest of cmp1[T] (max_red[T] returns T)

// редукция, возвращающая конкатенацию массивов или потоков
// или пустой массив или поток по умолчанию
type cat_red[T] := T
contract cat1[T]
  operation || (T, T returns T)
  function empty (returns T)
end contract
//$ identity
function catenate of cat1[T] (returns cat_red[T])
function catenate of cat1[T] (cat_red[T], T returns cat_red[T])
//$ associative
function catenate of cat1[T] (cat_red[T], cat_red[T] returns cat_red[T])
function catenate of cat1[T] (cat_red[T] returns T)
function empty of any[T] (returns array of T)
function empty of any[T] (returns stream of T)
```

6. ИНТЕРФЕЙС ВЗАИМОДЕЙСТВИЯ С ДРУГИМИ ЯЗЫКАМИ

Из программ на языке Sisal можно получать доступ к функциям на языках Си++, Си и любых других языков программирования, которые поддерживают использование своего кода из языка Си или Си++[57]. Из других языков программирования, которые поддерживают вызов внешних функций на языке Си, возможно использование необобщенных процедур языка Sisal.

[57] Например, язык Фортран-2003 поддерживает средства взаимодействия с языком Си.

6.1. Доступ к функциям языка Sisal из других языков

Конечной целью трансляции модуля на языке Sisal является единица компиляции на языке Си++, поэтому для использования процедур языка Sisal, объявленных в интерфейсе модуля, из программы на языке Си++ в ней достаточно подключить заголовочный файл «sisal.hpp», описать прототипы внешних функций, придерживаясь правил, описанных в разделе 9, скомпилировать и слинковать всё вместе.

Транслятор языка Sisal для обеспечения поддержки языка Си генерирует функции-оболочки на этом языке для всех процедур, объявленных в интерфейсе модуля. Обобщенные процедуры вызывать из программ на языке Си не разрешается[58]. Поддержка вызова процедур языка Sisal из программ на языке Фортран осуществляется средствами языка Фортран-2003, обеспечивающими вызов функций языка Си из программ на языке Фортран.

Имена функций на языке Си++ и языке Си совпадают. Пользовательские типы задаются встроенными типами, лежащими в их основе. Все аргументы функций языка Sisal передаются по значению. Функции языка Sisal, возвращающие несколько значений, возвращают их с помощью записи языка Си, поля которой имеют типы, соответствующие типам языка Sisal на языке Си, которые возвращаются функцией в естественном порядке их следования. Ниже приведено описание простых встроенных типов языка Си из заголовочного файла «sisal.h», задающих типы языка Sisal:

```
// пустой тип
enum SisalNull { nil };

// булевский тип
enum SisalBooleanType { False, True };
struct SisalBoolean {
  SisalBooleanType error[59];
  SisalBooleanType value;
};

// символьный тип
struct SisalCharacter {
  SisalBooleanType error;
  SisalCharacterType value;
};

// целый тип
struct SisalInteger {
```

[58] Это ограничение происходит из того, что обобщённые процедуры задаются шаблонами языка Си++, которые невозможно использовать с помощью средств языка Си.

[59] Флаг ошибки, равный *true*, если значение типа ошибочно, и *false* иначе.

```
  SisalBooleanType error;
  SisalIntegerType value;
};

// вещественный тип
struct SisalReal {
  SisalBooleanType error;
  SisalRealType value;
};
```

Составной тип записи языка Sisal на языке Си задаётся как структура, у которой первое поле, задающее флаг ошибочности записи, имеет тип SisalBooleanType, а типы остальных полей задают соответствующие типы полей записи языка Sisal на языке Си.

Составной тип объединения языка Sisal на языке Си задаётся как структура с двумя полями. Первое поле имеет тип *int* и содержит номер текущего тега объединения или отрицательное значение, если значение объединения ошибочно. Нумерация тегов ведётся с нуля, а сами теги упорядочены естественным порядком их задания в объединении языка Sisal. Второе поле содержит объединение с типами тегов объединения языка Sisal на языке Си в их естественном порядке следования.

Составной тип массива языка Sisal на языке Си задаётся как структура с четырьмя полями. Первое поле типа *int* содержит количество размерностей массива Dim (отсчитываемое с единицы) или значение меньше или равное нулю, если значение массива ошибочно. Второе поле с типом указателя на значение типа SisalIntegerType указывает на первый элемент массива с размерностью Dim, содержащего значения нижних границ размерностей, начиная с внешней размерности. Третье поле с типом указателя на значение типа SisalIntegerType указывает на первый элемент массива с размерностью Dim, содержащего значения верхних границ размерностей, начиная с внешней размерности. Четвёртое поле с типом указателя на элемент типа, соответствующего типу элемента массива языка Sisal на языке Си, указывает на первый элемент массива с размерностью, равной произведению количества элементов каждой размерности[60]. Элементы многомерного массива располагаются в row-major порядке. Памятью указателей входных (в функцию языка Sisal на языке Си) массивов нужно управлять самостоятельно. В обязанности вызывающего Си-кода входит освобождение памяти (функцией free), занимаемой указателями возвращаемых массивов.

[60] Количество элементов размерности массива равняется верхней границе размерности минус нижняя граница размерности плюс один.

Составной тип потока языка Sisal на языке Си задаётся как структура с шестью полями. Первое поле с типом указателя «*void**» указывает на некоторое состояние потока или задаёт ошибочное значение потока, если указатель равен нулю. Второе поле является указателем на функцию empty языка Sisal. Третье поле является указателем на функцию first языка Sisal. Четвёртое поле является указателем на функцию rest языка Sisal. Пятое поле является указателем на функцию, которая берёт и возвращает указатель «*void**», копируя состояние потока. Шестое поле является указателем на функцию, которая берёт указатель «*void**» и освобождает состояние потока им задаваемым. Памятью состояний входных потоков нужно управлять самостоятельно. В обязанности вызывающего Си-кода входит освобождение памяти, занимаемой состоянием возвращаемых потоков.

Тип функции языка Sisal на языке Си задаётся указателем на функцию, удовлетворяющую правилам описания функций языка Sisal на языке Си.

6.2. Интерфейс модуля на другом языке

Модули, реализованные на других языках программирования, называются инородными модулями. В интерфейсе инородного модуля указывается имя языка его реализации. Поддерживаются языки Си («C»), Си++ («Cpp») и Фортран («Fortran»). Содержимое интерфейса инородного модуля имеет специальный синтаксис.

Интерфейс инородного модуля может содержать объявления инородных процедур, определения переименованных и пользовательских типов и конструкции импорта переименованных и пользовательских типов других модулей.

Объявление инородной функции выглядит как «*function* имя функции (список формальных параметров *returns* тип возвращаемого значения)», где некоторые типы формальных параметров может предваряться ключевым словом «*in*», «*out*», «*in out*» или «*raw*». Ключевое слово «*out*» или «*raw*» может предварять тип возвращаемого значения. Использование любого из перечисленных ключевых слов возможно только перед множеством типов **S**, которому принадлежат инородные типы, пользовательские типы, в основании которых лежит тип из **S**, массивы от типа из **S**, записи с типами полей из **S** и объединения с типами тегов из **S** (рекурсивные зависимости объединений не допускаются). Далее при рассмотрении применимости этих ключевых слов, без дополнительных упоминаний, пользовательский тип рассматривается как встроенный тип, на котором он основан.

Ключевое слово «*raw*» может использоваться только перед типами записей и объединений. Ключевое слово «*raw*» перед типом записи означает, что в инородную процедуру передается (или для ключевого слова «*raw*» перед возвращаемым типом из инородной процедуры возвращается) структура с типами полей, соответствующих типам полей записи. Полям с типом записи или объединения, соответствуют поля типа «*raw*» запись и объединение. Полям типа массив со свободной формой, соответствует указатель в динамической памяти на последовательность всех его элементов (смотри описание способа передачи «*in* массива» ниже). Полям типа массив с фиксированной формой соответствует последовательность всех его элементов в записи. Ключевое слово «*raw*» перед типом объединения означает, что в инородную процедуру передается (или для ключевого слова «*raw*» перед возвращаемым типом из инородной процедуры возвращается) союз с типами полей, соответствующих типам тегов объединения таким же образом, как и в «*raw*» записи.

Ключевое слово «*in*» может использоваться перед всеми типами из **S**. Ключевое слово «*in*» означает, что в инородную процедуру передаётся указатель на некоторую выделенную динамическую память, содержащую копию значение типа. После вызова инородной процедуры копия значения типа и динамическая память, которую она занимает, освобождаются. Ключевое слово «*in*» перед массивом означает, что он задаётся указателем на последовательность всех его элементов. Элемент массива, являющийся массивом, задаётся указателем на выделенную динамическую память, содержащую последовательность всех его элементов. Элементы массивы, являющиеся записью или объединением, задаются непосредственно как «*raw*» записи или объединения, соответственно. Ключевое слово «*in*» перед записью или объединением означает, что они задаются указателем на «*raw*» записи или объединения, соответственно.

Ключевые слова «*in out*» могут использоваться перед типами множества $\mathbf{S_2}$, которому принадлежат инородные типы, массивы от типа из $\mathbf{S_2}$ и записи с типами полей из $\mathbf{S_2}$. Использование ключевых слов «*in out*» означает, что, как и для ключевого слова «*in*», в инородную процедуру передаётся указатель на динамическую память, только после вызова процедуры из копий значений в динамической памяти формируется возвращаемое значение типа формального параметра, которое считается дополнительным возвращаемым значением инородной процедуры. Размерности форм массивов считаются неизменными.

Ключевое слово «*out*» может использоваться перед типами из множества **S3**, которому принадлежат инородные типы, массивы с фиксированной

формой от типа из **S3**, записи с типами полей из **S3**. Для формального параметра с ключевым словом «*out*» в инородную процедуру передаётся (или для ключевого слова «*out*» перед возвращаемым типом из инородной процедуры возвращается) указатель на неинициализированный объект инородного типа или указатель на неинициализированную динамическую память с размером достаточным для хранения элементов массива с фиксированной формой или записи. После завершения работы инородной процедуры неинициализированная память считается содержащей возвращаемое значение, которое считается дополнительным возвращаемым значением инородной процедуры. Так же входное значение формального параметра с ключевым словом «*out*» не указывается, как если бы его не было.

Объявление инородной операции выглядит как «*operation* знак операции *is* имя функции (список формальных параметров *returns* тип возвращаемого значения)» или «*operation* знак операции *is* имя функции (список формальных параметров)», если список формальных параметров содержит хотя бы одно ключевое слово «*out*». Число и типы формальных параметров и возвращаемых значений операции должны удовлетворять требованиям, приведённым в табл. 1 с учётом действий ключевых слов «*in*» и «*out*».

В объявлении инородной процедуры можно не указывать тип возвращаемого значения вместе с ключевым словом «*returns*», если список формальных параметров содержит хотя бы одно ключевое слово «*out*»[61]. После объявления инородной процедуры можно указывать имя языка «*in* имя языка», если оно отличается от имени языка задаваемого интерфейсом инородного модуля, в котором оно содержится[62].

Объявление инородной процедуры может содержать прагму «weight = целочисленное выражение», где указанное целочисленное выражение задаёт значение веса, обозначающее приблизительное количество тактов, необходимых в среднем для исполнения данной процедуры. Для корректного сравнения весов инородных процедур различных модулей, полученных в разное время для разных архитектур, на уровне проекта программы можно задавать множитель, на который множатся все веса инородных процедур в указанном интерфейсе модуля. Прагма «weight» переопределяет значение веса в переобъявлении процедуры.

В интерфейсе инородного модуля на языке Си++ нельзя объявлять две процедуры с одинаковым именем (для инородной операции рассматривается её имя функции), одинаковой формой и разной формой возвращаемых

[61] Это требование позволяет сохранить хотя бы одно возвращаемое значение процедуры для языка Sisal.

[62] Например, в модуль на языке Си++ может экспортировать функции на языке Си++ и Си.

значений. Форма инородной процедуры включает признак передачи по указателю для каждого формального параметра, перед типом которого используются ключевые слова «*in*», «*out*» и «*in out*», и признак непосредственной передачи для каждого формального параметра, перед типом которого используются ключевое слово «*raw*». Формы инородных функций совпадают, если дополнительно совпадают указанные признаки типов формальных параметров.

В интерфейсе модуля на языке Си и Фортран не может быть объявлено более одной функции (включая имена функций операций) с одинаковым именем и разной формой. Модуль не может включать интерфейсы на языке Си и Фортран, которые содержат функции с одним именем и разной формой.

6.3. Предопределённые инородные типы

С компилятором языка Sisal для архитектуры x86 поставляется модуль с именем «Cpp», реализованный на языке Си++, который имеет следующий интерфейс на языке Sisal:

```
interface Cpp
  // тип указателя для возвращаемых массивов в динамической памяти (malloc)
  type ptr := "void*"
  contract any[T] end contract // на тип T нет никаких ограничений
  // конструирует массив из указателя, нижней и верхней границы массива,
  // освобождая (функцией free) память указателя;
  // если тип T не инородной тип или пользовательский тип,
  // в основе которого лежит инородной тип,
  // то возвращается ошибочное значение массива
  function make_array of any[T](ptr, integer, integer returns array of T)
  // запрещает копирование и запрещает объявлять и определять эту операцию
  no operation (ptr returns ptr)
  operation (null returns ptr) // error[ptr] это нулевой указатель

  // тип указателя для возвращаемых массивов в статической памяти
  type ptr_s := "void*"
  // конструирует массив из указателя;
  // если тип T не инородной тип или пользовательский тип,
  // в основе которого лежит инородной тип,
  // то возвращается ошибочное значение массива
  function make_array of any[T](ptr_s, integer, integer returns array of T)
  // запрещает копирование и запрещает объявлять и определять эту операцию
  no operation (ptr_s returns ptr_s)
  operation (null returns ptr_s) // error[ptr_s] это нулевой указатель

  // строки в динамической памяти, оканчивающиеся нулевым символом
  type psz := "char*"
  operation (array of character returns psz)
```

```
operation (psz returns array of character)
operation (psz returns psz)  // копирование строк возможно
operation (psz returns null)
operation (null returns psz) // error[psz] это нулевой указатель

// строки в статической памяти, оканчивающиеся нулевым символом
type psz_s := "char*"
operation (psz returns psz_s)
operation (psz_s returns psz)
operation (psz_s returns psz_s) // копируется указатель
operation (null returns psz) // error[psz_s] это нулевой указатель

// широкие строки в динамической памяти, оканчивающиеся нулевым символом
type pwsz := "wchar_t*"
operation (array of character returns pwsz)
operation (pwsz returns array of character)
operation (pwsz returns pwsz) // копирование широких строк возможно
operation (pwsz returns null)
operation (null returns pwsz) // error[pwsz] это нулевой указатель

// широкие строки в статической памяти, оканчивающиеся нулевым символом
type pwsz_s := "wchar_t*"
operation (pwsz returns pwsz_s)
operation (pwsz_s returns psz)
operation (pwsz_s returns pwsz_s) // копируется указатель
operation (null returns pwsz_s) // error[pwsz_s] это нулевой указатель

// булевский тип языка Си++
type bool = "bool"
operation (boolean returns bool)
operation (bool returns boolean)

// символьные типы языка Си и Си++
type char = "char"
type wchar = "wchar_t"
operation (character returns char)
operation (character returns wchar)
operation (char returns character)
operation (wchar returns character)

// архитектурно-зависимые типы целых чисел
type int8 = "__int8"
type int16 = "__int16"
type int32 = "__int32"
type int64 = "__int64"
operation (integer returns int8)
operation (integer returns int16)
operation (integer returns int32)
operation (integer returns int64)
operation (int8 returns integer)
operation (int16 returns integer)
```

```
  operation (int32 returns integer)
  operation (int64 returns integer)

  // архитектурно-зависимые типы вещественных чисел
  type real32 = "float"
  type real64 = "double"
  type real80 = "long double"
  operation (real returns real32)
  operation (real returns real64)
  operation (real returns real80)
  operation (real32 returns real)
  operation (real64 returns real)
  operation (real80 returns real)

  …  // другие операции инородных типов
end interface
```

7. СИНТАКСИЧЕСКАЯ И ЛЕКСИЧЕСКАЯ СТРУКТУРА ЯЗЫКА

В этом разделе приводится полное описание синтаксической и лексической структуры языка Sisal 3.2. Описание приводится в терминах грамматики ANTLR v3 [14] и лежит в классе LL_4 [14] языков.

Грамматика ANTLR содержит правила вида «нетерминал : альтернативы ;». Альтернативы разделяются знаком «|». Альтернатива состоит из (возможно пустой) последовательности термов, разделённых пробелами. Термом может быть нетерминал, строка терминалов, заключённых в одинарные кавычки, и альтернативы, заключённой в скобки. Терм, оканчивающийся знаком «?», является необязательным. Терм, оканчивающийся знаком «+», может повторяться один или более раз. Терм, оканчивающийся знаком «*», может повторяться ноль или более раз. Комментарии задаются как в языке Си++.

При описании лексической структуры текста используются дополнительные обозначения. Ключевое слово «fragment» перед правилом означает, что оно задаёт часть лексемы, используемую в другом правиле. Последовательности терминальных символов могут включать коды обратной косой черты. Можно задавать отрезки терминальных символов в виде «символ нижней границы отрезка .. символ верхней границы отрезка». Терм, задающий терминальные символы и предварённый знаком «~», обозначает все символы, кроме указанных.

7.1. Утверждения языка

Ниже приводится общая структура единицы компиляции программы:

```
compilation_unit:
  'module' mod_id (def_stmt ';'?)* 'end' 'module' |
  'interface' mod_id (decl_stmt ';'?)* 'end' 'interface' |
  'interface' mod_id 'in' lang_id (out_decl_stmt ';'?)* 'end' 'interface';

def_stmt:  def_import_stmt | type_def_decl | contract_def |
          'forward' func_decl | 'forward' op_decl | func_def | op_def;

decl_stmt: decl_import_stmt | type_def_decl | contract_def |
           func_decl | op_decl;

out_decl_stmt: type_def | out_func_decl | out_op_decl;
```

Далее указано устройства конструкций импорта:

```
decl_import_stmt: 'import' mod_id (',' mod_id)* | 'import' mod_id
                  (':' | '-') decl_import_obj (',' decl_import_obj)*;

decl_import_obj:  LexName | 'type' type_id | 'contract' contract_id;

def_import_stmt:  'import' mod_id (',' mod_id)* | 'import' mod_id
                  (':' | '-') def_import_obj (',' def_import_obj)*;

def_import_obj:
  LexName |
  'type' type_id |
  'contract' contract_id |
  'function' func_id '[' '..' ']' |
  'function' func_id of_param_names?
    ('[' out_types? ('returns' types)? ']') |
  'operation' op_cast of_param_names? '[' type 'returns' type ']' |
  'operation' op_1 of_param_names? '[' type ']' |
  'operation' op_12 of_param_names? '(' type (',' type)? ']' |
  'operation' op_2 of_param_names? '[' type ',' type ']' |
  'operation' op_func of_param_names? '[' types ']';

of_param_names: 'of' param_names;

param_names: '[' param_id (',' param_id)* ']';

op_all:  op_cast | op_1 | op_12 | op_2 | op_func;
op_cast: ':' |;
op_1:    '!' | '.' operation_id;
op_12:   '+' | '-';
op_2:    '*' | '/' | '%' | '**' | '^' | '&' |
         '|' | '||' | '[' ']' | '=' | '<';
op_func: '(' ')';
```

Описание структуры типов приводится ниже:

```
type_def_decl: type_def | type_decl;

type_def:  'type' type_id '=' type | 'type' type_id ':=' type;

type_decl: 'type' type_id param_names ':=' type;

type:  basic_type | foreign_type | type_ref | expr_type |
       array_type | stream_type | record_type | union_type | func_type;

types: type (',' type)*;

type_ref:    (mod_id '.')? type_id;
basic_type:  'null' | 'boolean' | 'character' | 'integer' | 'real';
foreign_type: LexStrConst;
expr_type:    'type' '(' expr ')';

array_type:     'array' (free_dim_form | fixed_dim_form) 'of' type;
free_dim_form:  '[' '..' (',' '..')* ']';
fixed_dim_form: '[' dim_duplet (',' dim_duplet)* ']';
dim_duplet:     expr? '..' expr;

stream_type: 'stream' 'of' type;

record_type: 'record' '[' field_spec (';' field_spec)* ';'? ']';
field_spec:  field_id (',' field_id)* ':' type;

union_type:  'union' '[' tag_spec (';' tag_spec)* ';'? ']';
tag_spec:    tag_id (',' tag_id)* (':' type | );

func_type:   'function' '(' types? 'returns' types ')';
```

Контракты определяются следующим образом:

```
contract_def: 'contract' contract_id param_names of_contract?
              (func_decl | op_decl)* 'end' 'contract';

of_contract:  'of' contract_id param_names;
```

Функции и операции объявляются и определяются следующим образом:

```
func_decl: 'function' func_id of_contract?
           '(' arg_decls? 'returns' types ')';

op_decl: 'no'? 'operation' (
      (op_cast | op_1) of_contract? '(' arg_decl 'returns' type ')' |
      op_12 of_contract? '(' arg_decl (',' arg_decl)? 'returns' type ')' |
      op_2 of_contract? '(' arg_decl ',' arg_decl 'returns' type ')' |
      op_func of_contract? '(' arg_decls 'returns' types ')'
    );

arg_decls: arg_decl (',' arg_decl)*;
arg_decl:  arg_def | type;
```

```
arg_defs:  arg_def (',' arg_def)*;
arg_def:   value_id ':' type;

func_def: 'function' func_id of_contract? '(' arg_defs? 'returns' types ')'
          exprs 'end' 'function';

op_def:   'operation' (
  (op_cast | op_1) of_contract? '(' arg_def 'returns' type ')' |
  op_12 of_contract? '(' arg_def (',' arg_def)? 'returns' type ')' |
  op_2 of_contract? '(' arg_def ',' arg_def 'returns' type ')' |
  op_func of_contract? '(' arg_defs 'returns' types ')'
)  exprs 'end' 'operation';
```

Иноязычные функции и операции объявляются, как указано ниже:

```
out_func_decl: 'function' func_id
               '(' out_args? ('returns' out_ret_types)? ')' ('in' lang_id)?;

out_op_decl:   'operation' op_all 'is' func_id
               '(' out_args ('returns' out_ret_types)? ')' ('in' lang_id)?;

out_args: out_arg (',' out_arg)*;
out_arg:  value_id ':' out_type | out_type;

out_types: out_type (',' out_type)*;
out_type:  ('raw' | 'in' | 'out' | 'in' 'out')? type;

out_ret_types: out_ret_type (',' out_ret_type)*;
out_ret_type:  ('raw' | 'out') type;
```

7.2. Выражения языка

Общий вид выражения приведён ниже (бинарные операции разбираются с учётом их приоритетов):

```
exprs: expr (',' expr)*;
expr:  concatenation;

concatenation:  disjunction ('||' disjunction)*;
disjunction:    xor ('|' xor)*;
xor:            conjunction ('^' conjunction)*;
conjunction:    equivalence ('&' equivalence)*;
equivalence:    relation ( ('=' | '!=') relation )*;
relation:       add_sub ( ('<' | '<=' | '>' | '>=') add_sub )*;
add_sub:        mul_div ( ('+' | '-') mul_div )*;
mul_div:       exponentiation ( ('*' | '/' | '%') exponentiation )*;
exponentiation: prefix_op ('**' exponentiation)?;

prefix_op:  ('+' | '-' | '!') prefix_op | postfix_op;

postfix_op: primary (
```

```
  '(' expr? (',' expr?)* ')' |
  '[' placement (':=' exprs (';' placed_elements)*)? ';'? ']' |
  'replace' '[' field_defs ']' |
  '.' LexName |
  'is' 'tag' tag_id |
  'is' 'error' |
  ':' safe_type |
  ':' '[' type ']'
)*;

safe_type: safe_type_ref | safe_array_type | safe_stream_type |
           basic_type | foreign_type | record_type | union_type |
           func_type | expr_type;

safe_type_ref:    mod_id '.' type_id;
safe_array_type:  'array' (free_dim_form | fixed_dim_form) 'of' safe_type;
safe_stream_type: 'stream' 'of' safe_type;
```

Синтаксис операндов выражения приведён ниже:

```
primary:  '(' expr ')' | 'old'? LexName | constant |
          union_gen | record_gen | stream_gen | array_gen |
          let_expr | if_expr | case_expr | for_expr;

constant: 'false' | 'true' | 'nil' | LexIntConst |
          LexRealConst | LexCharConst | LexStrConst |
          func_value | 'error' '[' type ']';

func_value:
  func_id '.' '[' types ']' |
  'function' func_id '[' '..' ']' |
  'function' func_id '[' out_types? ('returns' types)? ']' |
  'function' func_id '.' '[' types ']' '[' out_types? ']' |
  'function' func_id of_typed_params
    '[' out_types? ('returns' types)? ']' |
  'operation' op_cast of_typed_params? '[' type 'returns' type ']' |
  'operation' op_1 of_typed_params? '[' type ']' |
  'operation' op_12 of_typed_params? '(' type (',' type)? ']' |
  'operation' op_2 of_typed_params? '[' type ',' type ']' |
  'operation' op_func of_typed_params? '[' types ']' |
  'function' func_id? '(' arg_defs? 'returns' types ')'
    exprs 'end' 'function';

of_typed_params: 'of' '[' typed_param (',' typed_param)* ']';

typed_param: param_id '=' type;
```

Выражения конструирования объединений и записей указаны ниже:

```
union_gen:  'union' type '[' tag_id (':=' expr)? ']';
record_gen: 'record' type '[' (field_defs | ':=' exprs) ']' |
            'record' '[' field_defs ']';

field_defs: field_def (';' field_def)* ';'?;
field_def:  deep_field_name (',' deep_field_name)* ':=' exprs;
deep_field_name: field_id ('.' field_id)*;
```

Потоки конструируются, как указано ниже:

```
stream_gen: 'stream' 'of' type? '[' triplets ']';
triplets:  (triplet (',' triplet)*)?;
triplet:   expr triplet23? | triplet23;
triplet23: '..' expr? ('..' expr)?;
```

Синтаксис конструктора массива приведён ниже:

```
array_gen : ('array' 'of' type?)? '[' triplets ']' |
            'array' type '[' elements ']' |
            'array' named_fixed_dims ('of' type? | type) '[' elements ']' |
            'array' fixed_dim_names type '[' elements ']';

named_fixed_dims: '[' named_dim_duplet (',' named_dim_duplet)* ']';
named_dim_duplet: (dim_id 'in')? expr? '..' expr;

fixed_dim_names: '[' fixed_dim_name (',' fixed_dim_name)* ']';
fixed_dim_name:  dim_id | '..';

elements: ':=' exprs |
          placed_elements (';' placed_elements)*
          (';' ('else' ':=' exprs ';'?)?)?;
placed_elements: placement ':=' exprs;
placement:       placement_part (',' placement_part)*;
placement_part:  named_triplet ('dot' named_triplet)*;
named_triplet:   (dim_id 'in')? triplet;
```

Выражение «*let*» задаётся следующим образом:

```
let_expr:   'let' name_defs 'in' exprs 'end' 'let';
name_defs:  name_def (';' name_def)* ';'?;
name_def:   name_decls ':=' exprs;
name_decls: name_decl (',' name_decl)*;
name_decl:  value_id (':' type)?;
```

Синтаксис условных выражений приведён ниже:

```
if_expr: 'if' expr 'then' exprs ('elseif' expr 'then' exprs)*
         ('else' exprs)? 'end' 'if';

case_expr:     'case' expr ('of' pattern (',' pattern)* 'then' exprs)+
               ('else' exprs)? 'end' 'case' | case_tag_expr;

case_tag_expr: 'case' 'tag' expr ('of' tag_id (',' tag_id)* 'then' exprs)+
               ('else' exprs)? 'end' 'case';

pattern: expr ('..' expr?)? | '..' expr;
```

Циклические выражения описаны далее:

```
for_expr:    for_body for_test for_returns 'end' 'do' |
             while_test for_body for_returns 'end' 'while' |
             until_test for_body for_returns 'end' 'until' |
             'for' for_part for_returns 'end' 'for';
```

```
for_part:     for_test for_body |
              for_body for_test |
              range_gen (
                ';' name_defs (for_body for_test? | for_test for_body)? |
                ';' (for_body for_test? | for_test for_body) | ';'
              )?;

for_body:     'do' name_defs;

for_test:     while_test | until_test;

while_test:   'while' expr;

until_test:   'until' expr;

range_gen:    dot_gen ('cross' dot_gen)*;

dot_gen:      dim_gen ('dot' dim_gen)*;

dim_gen:      dim_id 'in' (expr ('at' dim_names | triplet23)? | triplet23);

dim_names:    dim_name (',' dim_name)*;

dim_name:     dim_id | '..';

for_returns: 'returns' reduction (';' reduction)* ';'?;

reduction:    ('stream' | 'array') 'of' expr filter? |
              'array' free_dim_form 'of' expr |
              'array' fixed_dim_form 'of' expr 'at' dim_names |
              (value_id | record_gen) ('(' exprs ')')? 'of' exprs filter?;

filter:       ('when' | 'unless') expr;
```

Далее приводится определение синонимов идентификатора:

```
mod_id:       LexName;
dim_id:       LexName;
param_id:     LexName;
operation_id: LexName;
contract_id:  LexName;
program_id:   LexName;
lang_id:      LexName;
func_id:      LexName;
type_id:      LexName;
tag_id:       LexName;
field_id:     LexName;
value_id:     LexName;
```

7.3. Лексическая структура языка

Текст модуля задан символами уникода (Unicode-16) [15] в UTF-8 [16] кодировке. Множество символов алфавита языка ограничено прописными и

строчными буквами латинского алфавита, арабскими цифрами и специальными символами, приведенными в таблице 4 вместе с их десятичными ASCII [12] кодами. Следующие символы называются пробельными и разделяют другие конструкции языка: табуляция (десятичный ASCII код 9), перевод строки (код 10), вертикальная табуляция (код 11), новая страница (код 12), возврат каретки (код 13) и пробел (код 32). Остальные символы, не принадлежащие алфавиту языка, могут входить только в состав комментариев, символьных и строковых литералов.

Таблица 4

Специальные символы

Знак	!	"	#	%	&	'	(	)	*	+	,	-	.	/
Код	33	34	35	37	38	39	40	41	42	43	44	45	46	47
Знак	:	;	<	=	>	@	[	\	]	^	_	{	|	}
Код	58	59	60	61	62	64	91	92	93	94	95	123	124	125

Ниже приводится формальное описание пробельных и нераспознаваемых символов:

```
// Whitespace is ignored
LexWhiteSpace: LexLineWhiteSpace | '\n';
fragment LexLineWhiteSpace: '\t' | '\000B' /*\v*/ | '\f' | '\r' | ' ';

// Unexpected character is ignored with warning.
LexUnknownChar: '\u0000'..'\u0008' | '\u000e'..'\u001f' |
                '#' | '$' | '?' | '@' | '\\' | '`' | '{' | '}' | '~' |
                '\u007F'..'\uFFFE';
```

Допустимы строковые комментарии, начинающиеся символами «//», и не вложенные друг в друга блочные (многострочные) комментарии, начинающиеся символами «/*» и оканчивающиеся символами «*/» (комментарий без завершения продолжается до конца файла). Строчный комментарий эквивалентен символу перевода строки, а блочный комментарий рассматривается как пробельный символ при трансляции. Комментарий, который начинается с символа доллара «$», называется прагмой и задаёт свойства конструкции, идущей следом (одной конструкции можно сопоставлять несколько прагм). Прагма может иметь вид «имя» или «имя = унарное выражение», где в унарном выражении могут принимать участие имена, видимые в месте расположения прагмы. Нераспознанные прагмы вызывают предупреждения компилятора. Ниже приводится описание комментариев и прагм:

```
LexLineCommentOrPragma: '/' '/' (
  // Pragma will be parsed later separately as "LexName ('=' expr)?".
  '$' ~'\n'* '\n'? |
  ~('$' | '\n') ~'\n'* '\n'? | '\n' // Comment is ignored.
);

LexBlockCommentOrPragma: '/' '*' (
  // Pragma will be parsed later separately as "LexName ('=' expr)?".
  '$' (options {greedy=false;}: .)* '*' '/' |
  // Comment is ignored.
  ~('$' | '*') (options {greedy=false;}: .)* '*' '/' |
  '*' ('/' | (options {greedy=false;}: .)* '*' '/')
);
```

Идентификаторы задаются цепочкой букв верхнего регистра и букв нижнего регистра (отличных от букв верхнего регистра), десятичных цифр и знака подчеркивания. Идентификатор не может начинаться с десятичной цифры и состоять из единственного знака подчеркивания. Далее идёт описание лексем идентификаторов и числовых литералов:

```
LexName:      LexLetter (LexLetter | LexDigit | '_')*;
LexDummyName: '_';
fragment LexLetter: 'A'..'Z' | 'a'..'z';
fragment LexDigit:  '0'..'9';

LexIntConst:  (LexDigit+ '#')? LexDigit+;

LexRealConst: LexDigit+ (LexRealExpField | '.' LexDigit* LexRealExpField?);
fragment LexRealExpField: ('E' | 'e') ('-' | '+')? LexIntConst;
```

Ниже приводится определение лексем символьных и строковых литералов:

```
LexCharConst: '\'' (
  (
    ( '\\' ( LexEscChar | LexEscCode |
       // Unrecognized escape-sequence.
       // Space character assumed.
       ~(LexEscChar | LexDigit | 'o' | 'h') |
       'o' ~LexOctDigit | 'h' ~LexHexDigit
         )
    ) | ~('\\' | '\'' | '\n')
  ) (
    '\'' |
    // A character constant can not contain not a single character.
    // Ignore extra characters until single quote or end of file.
    ~'\'' ~'\''* '\''?
  ) | (
    // A character constant can not be empty.
    // Space character assumed.
    '\'' |
```

```
    // A character constant may not extend across a line boundary.
    // Space character assumed.
    // Skipping until next single quote or end of file.
    '\n' ~'\''* '\''? |
    // A character constant may not contain end of file.
    // Space character assumed.
    EOF
  )
);

LexStrConst: (LexStr | LexRawStr)+;

fragment LexStr: '"' (
  // Semicolon after LexEscCode is ignored
  ( '\\' ( '"' | LexEscChar | LexEscCode (';' | ~('\\' | '"' | '\n' | ';')) |
     // Unrecognized escape-sequence.
     // Space character assumed.
     ~(LexEscChar | LexDigit | 'o' | 'h' | '"') |
     'o' ~LexOctDigit | 'h' ~LexHexDigit
               )
    ) | ~('\\' | '"' | '\n')
)* (
  '"' |
  // A character string constant may not extend across a line boundary.
  // End of string assumed.
  // Skipping until next double quote or end of file.
  '\n' ~'"'* '"'? |
  // A character string constant may not contain end of file.
  // End of string assumed.
  EOF
);

fragment LexRawStr: '@' '"' ( '\\' '"' | ~('\\' | '"' | '\n') )* (
  '"' |
  // A character string constant may not extend across a line boundary.
  // End of string assumed.
  // Skipping until next double quote or end of file.
  '\n' ~'"'* '"'? |
  // A character string constant may not contain end of file.
  // End of string assumed.
  EOF
);

fragment LexOctDigit: '0'..'7';
fragment LexHexDigit: '0'..'9' | 'A' .. 'F' | 'a' .. 'f';
fragment LexEscChar: '\'' | '\\' | 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v';
fragment LexEscCode: LexDigit+ | 'o' LexOctDigit+ | 'h' LexHexDigit+;
```

Ниже приводятся описание всех лексем, не определяемых правилами грамматики, а также список ключевых слов (42 штуки), которые не могут быть идентификаторами:

```
tokens {
  LexAnd   = '&';  LexAssign    = ':='; LexColon = ':';
```

```
  LexComma = ',';  LexConcat    = '||'; LexDiv   = '/';
  LexDot   = '.';  LexDots      = '..'; LexEQ    = '=';
  LexExp   = '**'; LexGE        = '>='; LexGT    = '>';
  LexLE    = '<='; LexLPar      = '(';  LexLSPar = '[';
  LexLT    = '<';  LexMinus     = '-';  LexMod   = '%';
  LexMul   = '*';  LexNE        = '!='; LexNOT   = '!';
  LexOr    = '|';  LexPlus      = '+';  LexRPar  = ')';
  LexRSPar = ']';  LexSemicolon = ';';  LexXOR   = '^';

  Lex_array     = 'array';     Lex_at       = 'at';
  Lex_case      = 'case';      Lex_contract = 'contract';
  Lex_cross     = 'cross';     Lex_do       = 'do';
  Lex_dot       = 'dot';       Lex_else     = 'else';
  Lex_elseif    = 'elseif';    Lex_end      = 'end';
  Lex_error     = 'error';     Lex_false    = 'false';
  Lex_for       = 'for';       Lex_forward  = 'forward';
  Lex_function  = 'function';  Lex_if       = 'if';
  Lex_import    = 'import';    Lex_in       = 'in';
  Lex_interface = 'interface'; Lex_is       = 'is';
  Lex_let       = 'let';       Lex_module   = 'module';
  Lex_nil       = 'nil';       Lex_no       = 'no';
  Lex_of        = 'of';        Lex_old      = 'old';
  Lex_operation = 'operation'; Lex_out      = 'out';
  Lex_raw       = 'raw';       Lex_record   = 'record';
  Lex_replace   = 'replace';   Lex_returns  = 'returns';
  Lex_stream    = 'stream';    Lex_tag      = 'tag';
  Lex_then      = 'then';      Lex_true     = 'true';
  Lex_type      = 'type';      Lex_union    = 'union';
  Lex_unless    = 'unless';    Lex_until    = 'until';
  Lex_when      = 'when';      Lex_while    = 'while';
}
```

7.4. Препроцессор

Транслятор языка Sisal 3.2 включает препроцессор, осуществляющий условную трансляцию, генерацию пользовательских предупреждений и ошибок, управление нумерацией строк и выделение именованных областей программы. В основе препроцессора лежит препроцессор языка C# [17]. Препроцессор языка является частью лексического анализатора и называется препроцессором для сохранения терминологии языков Си / Си++.

Каждая директива препроцессора занимает отдельную строку программы и начинается с символа «#», перед которым может стоять произвольное число пробельных символов. Далее сразу должно находиться имя директивы препроцессора:

- директивы «#define» и «#undef» — определение и отмена определения символов условной трансляции;
- директивы «#if», «#elseif», «#else» и «#endif» — условная трансляция секций текста программы;

- директива «#line» — управление нумерацией строк, использующейся в сообщениях об ошибках и предупреждениях;
- директивы «#error» и «#warning» — генерация пользовательских ошибок и предупреждений;
- директивы «#region» и «#endregion» — дополнительная пометка секций текста программы.

Директивы препроцессора, лежащие в многострочных блочных комментариях, игнорируются. К одной директиве препроцессора, кроме директив «#define», «#undef», «#else» и «#endif», относятся также и строки, идущие следом и начинающиеся символами «возможные пробельные символы # пробел». Содержимое последующих строк после символа «#» присоединяется к первой строке директивы.

Директива «#define имя» определяет имя со значением булевского литерала *true*, а директива «#undef имя» определяет имя со значением булевского литерала *false*. Значение определенного имени доступно только в директивах препроцессора, стоящих ниже по тексту. Значение неопределенного ранее имени равно булевскому литералу *false*. Не накладывается ограничений на любое переопределение указанных имен. Определения действуют до конца текущего файла.

Конструкция условной трансляции задаётся несколькими директивами:

```
#if булевское выражение
  секция текста программы
необязательные ветви #elseif
ветвь #else
#endif
```

Ветвь «#elseif» задаётся следующим образом:

```
#elseif булевское выражение
  секция текста программы
```

Ветвь «#else» задаётся следующим образом:

```
#else
  секция текста программы
```

Булевским выражением является любое булевское выражение языка Sisal, содержащее имена символов условной трансляции и литералы *true* и *false*. Первое истинное булевское выражение директив «#if» и «#elseif» определяет транслируемую обычным образом секцию текста программы, возможно тоже содержащую вложенные директивы условной трансляции. Если значения всех булевских выражений ложны, то транслируется секция текста программы, лежащая после директивы «#else», если она присутству-

ет. Транслируемая секция программы начинается со следующей после директивы строки или после окончания многострочного блочного комментария, начинающегося на одной строке с директивой препроцессора.

Пропускаемые секции текста программы не обязаны содержать лексически правильный текст. Их просмотр осуществляется только для корректного пропуска вложенных директив условной трансляции, не лежащих в многострочных блочных комментариях.

Директива «#line» имеет следующий синтаксис: «#line номер строки , имя файла», где номер строки и имя файла заданы выражениями языка Sisal целого и строкового типа. Директива меняет нумерацию и имя файла в возможных сообщениях об ошибках и предупреждениях текущего файла, начиная со следующей строки текста программы. Номер строки или имя файла (вместе с запятой) можно опустить для сохранения его предыдущего значения. Если опущен номер строки и номер файла, то восстанавливаются значения по умолчанию, как если бы не было ни одной директивы «#line» до этого.

Директива «#error сообщение об ошибке» генерирует ошибку трансляции с указанным сообщением, заданным выражением строкового типа языка Sisal. Сообщение об ошибке необязательно, и при его отсутствии будет сообщаться лишь о ее местоположении. Директива «#warning» аналогична директиве «#error» за исключением того, что вместо ошибки генерируется предупреждение.

Тексту можно прикрепить пользовательскую пометку директивами:

```
#region необязательная пометка
  секция текста программы
#endregion необязательная пометка
```

Необязательная пометка задается выражением строкового типа языка Sisal и не обязана совпадать в директивах «#region» и «#endregion». Секция текста программы транслируется обычным образом и также может содержать вложенные директивы «#region» … «#endregion».

Ниже схематически приведён разбор директив препроцессора на этапе лексического анализа теста:

```
// Preprocessor directives are parsed
// after lexical analysis and before syntax analysis

PreDirDefUndef: PreDirBegin ('define' | 'undef')
                LexLineWhiteSpace+ LexName PreDirEnd;

// Directives accumulate string that will be parsed later separately
PreDirCont:  PreDirBegin
             ('if' | 'line' | 'error' | 'warning' | 'region' | 'endregion')
```

```
            ~'\n'* '\n'? (PreDirBegin LexLineWhiteSpace ~'\n'* '\n'?)*;

PreDirSimple: PreDirBegin  ('else' | 'endif') PreDirEnd;

fragment PreDirBegin: {getColumn()==1}? LexLineWhiteSpace* '#';

fragment PreDirEnd: LexLineWhiteSpace*
  (LexLineCommentOrPragma | LexBlockCommentOrPragma | '\n');
```

8. ФОРМАТ FIBRE ДЛЯ ВНЕШНИХ ЗНАЧЕНИЙ

Формат Fibre описывает синтаксис языка взаимодействия с функцией main на языке Sisal. Формат Fibre описывает текстовое представление типов языка Sisal, которые могут использоваться функцией «main» на языке Sisal. Описание приводится в терминах грамматики ANTLR v3, как и в разделе 7, и лежит в классе LL_1 [14] языков (описание неопределённых здесь лексем находится в разделе 7.3).

```
tokens {
  LexColon = ':';  LexComma = ',';
  LexDots  = '..'; LexGT    = '>';
  LexLCPar = '{';  LexLPar  = '(';
  LexLSPar = '[';  LexLT    = '<';
  LexMinus = '-';  LexNE    = '!=';
  LexPlus  = '+';  LexRCPar = '}';
  LexRPar  = ')';  LexRSPar = ']';

  Lex_error = 'error';  Lex_false = 'false';
  Lex_nil   = 'nil';    Lex_true  = 'true';
}

fibre_unit: (value)*;

value: simple_value | compound_value | 'error';

simple_value:   ('+' | '-') (LexIntConst | LexRealConst) | LexCharConst |
                 'false' | 'true' | 'nil';

compound_value: array | stream | record | union;

array:  '[' ( range+ ':' value+  )? ']' | LexStr;

range:  LexIntConst '..' LexIntConst;

stream: '{' value* '}';

record: '<' value+ '>';

union:  '(' LexIntConst ':' value ')';
```

```
// Whitespace is ignored
LexWhiteSpace:   '\t' | '\n' | '\000B' /*\v*/ | '\f' | '\r' | ' ';

// Comments are ignored.
LexLineComment:  '/' '/' ~'\n'* '\n'?;
LexBlockComment: '/' '*' (options {greedy=false;}: .)* '*' '/';
```

Формат Fibre достаточно очевиден и не требует особых пояснений. Ключевые слова чувствительны к регистру. Ключевое слово «*error*» задаёт ошибочное значение любого типа. Число в представлении объединения обозначает порядковый номер (начинающийся с единицы) задаваемого тега. Значения элементов массива перечисляются в row-major порядке. Не заданные элемента массива и записи предполагаются равными ошибочным значениям (выводится предупреждение). Лишние элементы массива и записи игнорируются и вызывают предупреждение.

9. СТРУКТУРА МОДУЛЯ НА ЯЗЫКЕ СИ++

Конечной целью трансляции модуля на языке Sisal является единица компиляции на языке Си++, удовлетворяющая определённым правилам, описанным в данном разделе.

9.1. Заголовочный файл «sisal.hpp»

В подключаемом заголовочном файле «sisal.hpp» описываются типы языка Sisal на языке Си++, заданные с помощью шаблонов классов. Все объявления заголовочного файла «sisal.hpp» расположены в пространстве имён «Sisal».

Для простых встроенных типов языка Sisal на языке Си++ определяется класс пустого типа Null, булевского типа Boolean, символьного типа Character, целого типа Integer и вещественного типа Real. Данные классы (кроме класса Null) определяют методы error и value. Метод error возвращает значение типа *bool*, равное значению *true*, если значение языка Sisal является ошибочным, и *false* иначе. Метод value возвращает значение *bool* для класса Boolean, значение SisalCharacterType[63] для класса Character, значение SisalIntegerType[64] для класса Integer и значение SisalRealType[65] для класса Real.

[63] Тип SisalCharacterType определяется как *typedef* от некоторого символьного типа языка Си (скорей всего от типа wchar_t).

[64] Тип SisalIntegerType определяется как *typedef* от некоторого целого типа со знаком языка Си (скорей всего от типа *int*).

Встроенные составные типы языка Sisal определяются через шаблоны классов языка Си++. Для классов, соответствующих встроенным типам языка Sisal, в заголовочном файле определены все встроенные операции.

Тип потока задаётся шаблоном класса «Stream<**T**>», где параметр шаблона **T** задаёт тип элементов потока. Тип массива со свободной формой задаётся шаблоном класса «Array**N**<**T**>», где число **N** является частью имени шаблона и задаёт размерность массива, а параметр шаблона **T** задаёт тип элементов массива. Тип массива с фиксированной формой задаётся шаблоном «Array_$\mathbf{D_1}$_$\mathbf{D_2}$_..._$\mathbf{D_N}$<**T**>», где числа $\mathbf{D_1}$, …, $\mathbf{D_N}$ задают количество элементов в соответствующих размерностях массива. Шаблоны класса массива поддерживают операцию получения элемента по его возможно многомерному индексу. Объявить тип массива со свободной формой можно макросом «SISAL_DECLARE_ARRAY(**T**, **N**)». Объявить тип массива с фиксированной формой можно макросом «SISAL_DECLARE_FIXED_ARRAY(**T**, $\mathbf{D_1}$, …, $\mathbf{D_N}$)», где число **N** в текущей реализации ограничено[66] числом 100.

Тип записи задаётся шаблоном класса «Record<$\mathbf{T_1}$, …, $\mathbf{T_n}$[67]>», где типы $\mathbf{T_1}$, …, $\mathbf{T_n}$ задают типы полей записи. Тип объединения задаётся шаблоном класса «Union<$\mathbf{T_1}$, …, $\mathbf{T_n}$>», где типы $\mathbf{T_1}$, …, $\mathbf{T_n}$ задают типы тегов объединения. Доступ к полям записи и тегам союза осуществляется, соответственно, с помощью методов GetField**i** и GetTag**i**, где число **i** лежит в отрезке от **0** до **n-1**.

Тип функции задаётся шаблоном класса «Function<Tuple<$\mathbf{A_1}$, …, $\mathbf{A_n}$>, Tuple<$\mathbf{R_1}$, …, $\mathbf{R_m}$> >», где типы $\mathbf{A_1}$, …, $\mathbf{A_n}$ задают типы формальных параметров функции, а типы $\mathbf{R_1}$, …, $\mathbf{R_m}$ задают типы возвращаемых значений. Шаблон класса «Tuple<$\mathbf{T_1}$, …, $\mathbf{T_n}$>» используется для задания кортежей значений, доступ к элементам которых предоставляет метод GetItem**i**, где число **i** лежит в отрезке от **0** до **n-1**. Шаблон класса функции поддерживает операцию вызова функции.

---

[65] Тип SisalRealType определяется как *typedef* от некоторого вещественного типа языка Си (скорей всего от типа *double*).

[66] В силу невозможности задания макроса с переменным количеством параметров, необходимо было остановиться на каком-то числе.

[67] Число **n**, в силу ограничений реализации на количество параметров шаблона в некоторых компиляторах языка Си++ может не превосходить числа 26. В любом случае, в силу невозможности задания шаблона с переменным количеством параметров, необходимо было остановиться на каком-то числе.

9.2. Определения и объявления типов

Определения переименованных типов языка Sisal соответствуют *typedef* объявлениям типов на языке Си++ со строкой «s_» в начале имени переименованного типа. Определения пользовательских типов соответствуют определениям классов со строкой «s_» в начале имени пользовательского типа, наследуемых от класса базового типа. Объявления пользовательских типов с параметрами соответствуют определениям шаблонов класса со строкой «s_» в начале имени пользовательского типа с параметрами, наследуемыми от класса базового типа.

9.3. Определения и объявления процедур

Объявлениям и определениям функций языка Sisal соответствуют объявления и определения функций языка Си++ со строкой «s_» в начале имени и суффиксом, призванным отличать функции неоднозначные по типам возвращаемых значений. Объявлениям и определениям операций языка Sisal соответствуют объявления и определения операций языка Си++ с теми же знаками, за исключением операций со знаками «**», «. имя» и операций преобразования типов. Операции со знаком «**» соответствует функция с именем «op_pow». Операции со знаком «. имя» соответствует метод с указанным именем в классе, соответствующем пользовательскому типу, являющемуся типом формального аргумента операции. Операции явного преобразования типов соответствует функция с именем «op_cast». Операции неявного преобразования типов, результатом которой является пользовательский тип, соответствует конструктор класса, соответствующего указанному пользовательскому типу. Операции неявного преобразования типов, результатом которой не является пользовательский тип, соответствует операция преобразования в этот тип, расположенная в классе, являющимся типом формального аргумента операции.

Объявлениям и определениям обобщенных процедур соответствуют шаблоны функций (со строкой «s_» в начале имени) и операций (с учетом указанных выше исключений). Функции и операции языка Си++, задающие процедуры языка Sisal, берут на вход и возвращают шаблон класса «Tuple<$\mathbf{T_1}$, …, $\mathbf{T_n}$>».

СПИСОК ЛИТЕРАТУРЫ

1. **McGraw J. R.** Sisal: Streams and iterations in a single assignment language, Language Reference Manual, Version 1.2. / McGraw J. R., Skedzielewski S. K., Allan S. J., Oldehoeft R. R., Glauert J., Kirkham C., Noyce B. and Thomas R. — Livermore, CA, 1985. — (Tech. Rep. / Lawrence Livermore National Laboratory; M-146, Rev. 1).
2. **Стасенко А. П.** Транслирующие компоненты системы функционального программирования SFP / Глуханков М. П., Дортман П. А., Павлов А. А. и Стасенко А. П. // Современные проблемы конструирования программ. — Новосибирск, 2002. — С. 69–87.
3. **Cann D. C.** Retire Fortran?: a debate rekindled // Communs. of the ACM. — New York: ACM Press, 1992. — Vol. 35, No. 8. — P. 81–89.
4. **Cann D. C.** Sisal Reference Manual: Language Version 2.0 / Cann D. C., Feo J. T., Böhm A. P. W. and Oldehoeft R. R. — Livermore, CA, 1991. — 128 p. — (Tech. Rep. / Lawrence Livermore National Laboratory; UCRL-MA-109098).
5. **Feo J. T.** Sisal 90 user's guide / Feo J. T., Miller P. J., Skedzielewski S. K. and Denton S. M. — Livermore, CA: Lawrence Livermore National Laboratory, Draft 0.96, 1995. — 80 p.
6. **Бирюкова Ю. В.** Sisal 90: Руководство для пользователя. — Новосибирск, 2000. — 84 с. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 72).
7. **ISO 7185:1990(E).** Information technology: Programming languages: Pascal. — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 1990.
8. **Касьянов В. Н.**, **Бирюкова Ю. В.** и **Евстигнеев В. А.** Функциональный язык Sisal // Поддержка супервычислений и интернет-ориентированные технологии. — Новосибирск, 2001. — С. 54–67.
9. **Стасенко А. П.**, **Синяков А. И.** Базовые средства языка Sisal 3.1. — Новосибирск, 2006. — 60 с. — (Препр. / РАН. Сиб. отд-ние. ИСИ; № 132).
10. **Стасенко А. П.** Внутреннее представление системы функционального программирования Sisal 3.0. — Новосибирск, 2004. — 54 с. — (Препр. / РАН. Сиб. отд-ние. ИСИ; № 110).
11. **ISO/IEC 14882:2003(E).** Programming languages: C++. — Geneva: International Organization for Standardization (ISO), Central Secretariat, 2003.
12. **ANSI X3.4:1986.** Information systems: coded character sets: 7-Bit American national Standard Code for Information Interchange (7-Bit ASCII). — NY: American National Standards Institute (ANSI), 1986.
13. **ANSI/IEEE 754-1985.** IEEE standard for binary floating-point arithmetic. — NY: Institute of Electrical and Electronics Engineers, 1985 (Reprinted in SIGPLAN Notices, 22(2): 9–25, 1987).
14. **Parr T.** The Complete Antlr Reference Guide. — Pragmatic Bookshelf, 2007. — 361 p.
15. **The Unicode Consortium.** The Unicode Standard. — Version 4.0. — Boston, MA: Addison-Wesley, 2003.

16. **ISO/IEC 10646:2003(E).** Information technology: Universal Multiple-Octet Coded Character Set (UCS). — Geneva: International Organization for Standardization (ISO), Central Secretariat, 2003.
17. **Робинсон У.** C# без лишних слов / Пер. с англ. — М.: ДМК Пресс, 2002. — 352 с.: ил. (Серия «Для программистов»).