**Российская академия наук**
**Сибирское отделение**
**Институт систем информатики**
**им. А. П. Ершова**

**Юлия Владиславовна Бирюкова**

# SISAL 90
# РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

**Препринт**
**72**

**Новосибирск 2000**

В работе представлено описание языка Sisal 90. Он содержит черты современных функциональных и императивных языков. Добавление некоторых возможностей языка Фортран 90 и поддержка смешанно-языкового программирования повысило пригодность Sisal 90 для научного программирования.

Данная работа выполнена в рамках проекта «Методы и средства функционального программирования поддержки супервычислений» (грант РФФИ № 98-01-748).

**Siberian Division of the Russian Academy of Sciences**
**A. P. Ershov Institute of Informatics Systems**

**Yulia V. Biryukova**

# SISAL 90
# USER TUTORIAL

**Preprint**
**72**

**Novosibirsk 2000**

In this writing the Sisal 90 language definition is presented. Sisal 90 supports modern functional and imperative language features. It increases the language's utility for scientific programming by adding features from Fortran 90 and support for mixed language programming.

## ПРЕДИСЛОВИЕ

Sisal (Streams and Iteration in a Single Assignment Language) является языком функционального программирования, ориентированым на параллельную обработку. Его можно назвать потомком языков VAL и Id [1], хотя он содержит некоторые черты, не присущие ни одному из них [2-4]. Язык был разработан в 1983 году совместными усилиями Ливерморской национальной лаборатории имени Лоренца, университетов Колорадо и Манчестера и Digital Equipment Corporation (DEC). В 1985 г. была зафиксирована версия Sisal 1.2 [5]. Его можно охарактеризовать как функциональный язык, поддерживающий типы данных и операций для научных расчетов. Какое-то время его даже рассматривали как замену языка Фортран. В 1991 году Ливерморская лаборатория и университет Колорадо опубликовали версию языка Sisal 2.0 [6]. Эта версия содержала в себе такие идеи современных функциональных языков, как функции высшего порядка, конструкцию type set и, наконец, модули, ее краткое описание можно найти в [7]. В 1995 году вышла в свет версия Sisal 90, которой посвящена данная работа.

С точки зрения семантики, язык обладает рядом важных свойств. Во-первых, он математически правильный, то есть функции отображают входы в выходы без сторонних эффектов. В последней версии это свойство нарушено введением глобальных значений, которые могут поступать в функцию извне и не указываются как ее параметры. Во-вторых, имена прозрачны для ссылок, то есть они олицетворяют значения, а не ячейки памяти. В-третьих, Sisal — это язык однократного присваивания.

Исполняемая единица языка состоит из одной или нескольких функций, среди которых обязательно присутствует функция с именем main, являющаяся точкой входа в программу.

Sisal поддерживает стандартные скалярные типы данных: булевские, целые, действительные одинарной и двойной точности, комплексные одинарной и двойной точности и null, из структурных типов имеются массивы, потоки, записи, союзы и type set. Для определения сложной структуры данных необходимо указать типы составляющих ее элементов, число элементов при этом не указывается и становится известным только при присвоении структуре значения. Массивы и потоки состоят из однотипных элементов и различаются прямым и последовательным доступом к элементам. Записи и союзы состоят из разнотипных элементов и различаются способом хранения.

Основной синтаксической единицей языка является выражение, оно олицетворяет список значений некоторых типов. Арность выражения — это

размер списка значений. Основным выражением или конструкцией языка является let-выражение, оно вводит имена значений и заголовки функций, производит над ними некоторые действия и выдает результаты.

В языке присутствуют все виды циклов: итерационные с инициализацией значений, с пред- и пост-условием, а также явные параллельные циклы, экземпляры тела которого могут параллельно выполняться на различных процессорах.

К выражениям выбора относятся конструкции if и case, в последней добавилась возможность выбирать ветвь не только по значению, но и по типу имени.

Функция языка Sisal состоит из заголовка и тела. В заголовке объявляются имена и типы формальных параметров и типы возвращаемых результатов. Имеется возможность определения функций вперед и создания рекурсивных функций, а также описания функций, закодированных на других языках программирования.

Версия Sisal 90 включает в себя векторные операции над массивами, семантика которых аналогична семантике векторных операций языка Фортран 90. Такие операции работают по простой рекурсивной схеме.

Версия Sisal 1.2 реализована на следующих машинах: Denelcor HEP, Vax 11-780, Cray-1, Cray-X/MP, Prototype Dataflow Computer [4], Ncube 2000 и Intel Paragon [8]. Для нее имеется работающий оптимизирующий транслятор OSC. Предполагается, что язык Sisal содержит набор прагм, с помощью которых можно управлять процессом компиляции путем явного задания параметров этого процесса. Например, при помощи прагм можно указывать характеристики целевой архитектуры и выделять подмножества языка Sisal для реализации.

В основу данной работы лег препринт [9].

## 1. ОСНОВНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА

При написании программ на языке Sisal 90 используется множество символов ASCII, из управляющих символов допустимы только tabs и end-of-line.

### 1.1. Имена

Имена представляют значения, типы и функции. Нельзя ссылаться на имя до его определения или, в случае функционального имени, до его декларации. Так как язык Sisal 90 является языком однократного присваива-

ния, в области может присутствовать только одно определение имени, однако можно использовать то же самое имя в другой, возможно вложенной, области, не опасаясь двусмысленности. Имя может иметь любую длину, размер литер в имени не существенен.

Литера подчерка (_) используется в различных целях. Если она стоит внутри имени, она обрабатывается как любая литера алфавита, например, compound_name и compoundname — два различных имени. Когда эта литера стоит в качестве действительного параметра в функциональном вызове, она обозначает место неспецифицированного параметра.

```
f := g ( _, 3)
```

Данное предложение языка определяет новую функцию с одним параметром. Эту функции представляет граф функции g, где первый параметр неспецифицирован, а второй имеет значение 3. Когда литера подчерка находится в левой части оператора присваивания, она констатирует, что значение, возвращаемое в этой позиции, можно отбросить за ненадобностью. Например, оператор присваивания

```
a, _ := integer_divide(5, 3)
```

инструктирует, что нужно первый результат вернуть, а второй отбросить.

Определенные имена зарезервированы для специальных целей и не могут использоваться для обозначения значений, типов или функций в программе. Далее перечислены зарезервированные слова в языке Sisal 90:

| | | | | |
|---|---|---|---|---|
| array | elseif | in | reduction | then |
| at | end | initial | repeat | type |
| case | for | is | replace | union |
| cross | forward | let | returns | unless |
| define | function | of | set | when |
| dot | global | old | stream | while |
| else | if | record | tag | |

Имена типов, имена операторов и большинство имен редукций являются предопределенными, а не зарезервированными, т.е. имеют первоначальный смысл, который при желании можно изменить.

## 1.2. Комментарии

Комментарий начинается знаком % и заканчивается на следующей литере конца строки. Язык поддерживает как полнострочные, так и встроенные комментарии.

## 1.3. Литералы

Sisal 90 поддерживает литералы семи типов: булевские (boolean), null, целые (integer), действительные (real), двойной точности (double), символьные (character) и строковые (string). Дополнительно программисты могут сконструировать значения комплексные (complex) и комплексные двойной точности (double_complex) из скалярных литералов.

**Булевские.** Булевскими литералами являются зарезервированные слова true и false.

**Null.** Зарезервированное слово nil является литералом типа null.

**Целые.** Целым литералом является непрерывная строка десятичных цифр. Диапазон целых литералов зависит от машины. Пробелы несут синтаксическую нагрузку, например, 1 000 000 интерпретируется как три литерала. Для упрощения недесятичных вычислений Sisal 90 поддерживает недесятичные целые. Их общая форма такова:

*основание # цифры,*

где *основание* есть любое целое от 2 до 16, а *цифры* — это целый литерал подходящего диапазона. Буквы от a до f (заглавные и прописные) используются для представления чисел от 10 до 15 соответственно. Ниже приведены примеры целых литералов.

123, +123, -123, 16#7b, 8#173, 2#1111011

**Действительные литералы.** Действительные литералы могут быть представлены либо в стиле с плавающей запятой, либо в экспоненциальной нотации.

Первый стиль задает следующие правила записи литералов:

- одна или более десятичных цифр, за которыми следует период, за которым следует ноль или более десятичных цифр, или
- период, за которым следует одна или более десятичных цифр.

Второй стиль предлагает несколько иную запись литералов:

- десятичное целое, за которым следует экспонента, или

- действительное с фиксированной точкой, за которым следует экспонента.

Под экспонентой понимается литера e или E, за которой следует десятичное целое возможно со знаком. Перед любым действительным литералом может присутствовать знак. Диапазон экспоненты и точность являются машинно-зависимыми характеристиками. Далее приведены примеры действительных литералов.

123., .123, 123.123, 123.1e5, 123.1E-5, 1e3, .123e0

**Литералы двойной точности.** Литералы двойной точности подобны действительным литералам, за исключением того, что поле экспоненты обязательно и начинается с символа d или D. Как диапазон экспоненты, так и точность литералов двойной точности не меньше, чем диапазон и точность действительных литералов, причем снова обе характеристики машинно-зависимы. Ниже приведены примеры литералов двойной точности.

.123d0, 123.d0, 123.1d5, 123.1d-5, 123D0, 123.d

**Комплексные литералы.** Комплексный литерал не является литералом как таковым, он есть результат предопределенной конструирующей функции *complex*. Функция воспринимает два численных значения — действительную и мнимую части — и возвращает комплексное значение одинарной точности.

*complex(2, 3.0), complex(2.1, 3.1)*

Каждое численное значение может быть целым, действительным или двойной точности. Целые значения и значения двойной точности приводятся к своему действительному эквиваленту.

**Комплексные литералы двойной точности.** Комплексные литералы двойной точности конструируются подобно комплексным литералам функцией *double_complex.*

*double_complex(2, 3.0), double_complex(2.1, 3.1)*

Каждое численное значение может быть целым, действительным или двойной точности. Целые и действительные значения приводятся к своему действительному эквиваленту двойной точности.

**Символьные литералы.** Символьным литералом является любой символ ASCII или управляющий символ, заключенный в апострофы. Непечатные символы, такие как табулятор или новая строка, а также символ апост-

рофа представляются последовательностью символов, начинающейся с обратной косой черты.

| | | |
|---|---|---|
| \n | новая строка | применение различно |
| \r | возврат каретки | ASCII CR |
| \t | горизонтальный табулятор | ASCII HT |
| \f | подача форматированных бланков | ASCII FF |
| \b | backspace | ASCII BS |
| \\ | обратная косая черта | ASCII \ |
| \” | двойная кавычка | для использования в строке |
| \’ | апостроф | для использования в символьных литералах |

Дополнительно можно специфицировать любой символ с помощью обратной косой черты, за которой следует восьмеричный код ASCII, а также с помощью обратной косой черты, за которой следует сам символ. Ниже даны примеры символьных литералов.

```
‘A’     ‘\A’     % буква A
‘\’     ‘\174’   % апостроф
‘\n     ‘\012’   % новая строка
‘\000’           % ASCII NULL
```

**Строки.** Строкой является ноль или более ASCII символов, заключенных в двойные кавычки.

“Hello world.\n”

Строка есть в действительности массив символов с нижней границей 1. Для включения в строку двойных кавычек нужно использовать последовательность символов \”.

“Jane said, \”See Spot run.\””

### 1.4. Операции

В языке Sisal 90 имеется четыре класса операций. В порядке возрастания приоритетов — арифметические, сравнения, булевские и строковые.

**Арифметические операции.** Арифметическими операциями в порядке возрастания приоритетов являются

```
-    +        унарный минус,  унарная идентичность,
```

| | |
|---|---|
| ** | возведение в степень, |
| * / mod | умножение, деление, деление по модулю, |
| + - | сложение, вычитание. |

Эти операции определены для всех численных операндов, включая комплексные и комплексные двойной точности. Как в языке Fortran, операции являются инфиксными операторами. Следует отметить, что mod поддерживается как инфиксный оператор и как функция. За исключением экспоненты все операции приводят неэквивалентные входные типы в соответствии с иерархией типов, изображенной в главе 1.5, и возвращают результат приведенного типа. Например, если целое значение умножается на действительное, сначала первый операнд приводится к действительному типу, и затем применяется оператор умножения. Результатом будет действительное значение.

Последовательность идентичных операций вычисляется слева направо, за исключением последовательности операций возведения в степень, которая вычисляется справа налево. Левые и правые части следующих арифметических выражений эквивалентны:

```
x1 * y1 + x2 * y2 = (x1 * y1) + (x2 * y2),
x + y + z = (x + y) + z,
x ** -y = x ** (-y),
i + j mod 4 = i + (j mod 4),
y ** x ** 2 = y ** (x ** 2).
```

Для изменения порядка вычислений можно использовать круглые скобки. Операции внутри скобок всегда вычисляются первыми, во вложенных скобках первыми вычисляются операции в самых внутренних скобках. Если a = 2, b = 3 и c = 4, тогда a * (b + (c + c) / a = 21.

**Операции сравнения.** Все операции сравнения перечислены ниже.

| | | | |
|---|---|---|---|
| = | равно | ~= | не равно |
| < | меньше | <= | меньше или равно |
| > | больше | >= | больше или равно |

Все операции сравнения имеют одинаковый приоритет и неассоциативны. Более того, последовательность операторов сравнения интерпретируются как конъюнкция операторов сравнения, как показано в нижеследующих выражениях:

$$(i < j < k) = ((i < j) \ \& \ (j < k)),$$
$$(i = j = k) = ((i = j) \ \& \ (j = k)),$$
$$(0 < i = j <= 99) = ((0 < i) \ \& \ (i = j) \ \& \ (j <= 99)).$$

Последовательность может быть произвольной длины и вычисляется слева направо.

**Булевские операции.** Sisal 90 поддерживает три булевские операции, они перечислены ниже в порядке уменьшения приоритетов.

| | |
|---|---|
| ~ | не |
| & | и |
| \| | или |

Последовательность булевских операций одинакового приоритета вычисляется слева направо. Левые и правые части следующих булевских выражений эквивалентны:

$$b1 \mid \tilde{}b2 \ \& \ b3 = b1 \mid ((\tilde{}b2) \ \& \ b3),$$
$$b1 \mid b2 \ \& \ b3 \mid b4 = (b1 \mid (b2 \ \& \ b3)) \mid b4,$$
$$b1 \ \& \ b2 \ \& \ b3 = (b1 \ \& \ b2) \ \& \ b3.$$

Булевские выражения вычисляются слева направо, и процесс заканчивается, как только значение выражения станет известным. Булевские выражения так называемого “короткого замыкания” полезны при написании тестов завершения для циклических выражений, предотвращающие деление на ноль или выход за границы массива. Например, булевское выражение (b ~= 0 & a/b > 0) является выражением короткого замыкания, если b = 0, исключая деление на ноль во втором терме. Если некоторый терм в булевском выражении является ошибочным (см. главу 1.6 для знакомства с ошибочными значениями), выражение возвращает ошибку, за исключением специальных случаев:

1. error[boolean] & false = false.
2. error[boolean] | true = true.

**Строковые операции.** Строковыми операциями являются конкатенация, извлечение символа и извлечение подстроки. Так как строки представляют собой массивы символов с нижней границей 1, индивидуальный символ извлекается по индексу, а извлечение подстроки — это то же самое, что извлечение подмассива (см. главу 4.1.4):

```
str || “\n”          % конкатенация
str[i]               % извлечение символа
str[i:j]             % извлечение подстроки
```

Символьное значение не идентично массиву символов, или строке длины 1.

```
let c := ‘a’
    in array[c],     % строка “a”
    [‘b’]            % строка “b”
end let
```

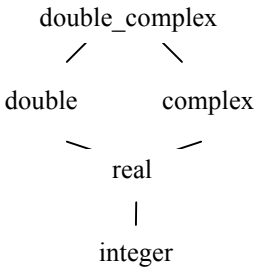
### 1.5. Операнды и их типы

Все операторы, описанные в предыдущем разделе, за исключением отрицания и унарного минуса, работают с двумя операндами. В следующей таблице даны правильные входные и выходные типы для каждого оператора.

| Оператор | Типы данных |
|---|---|
| - (унарный минус) | целые, действительные, двойной точности, комплексные, комплексные двойной точности |
| mod | целые, действительные, двойной точности |
| + - * / | целые, действительные, двойной точности, комплексные, комплексные двойной точности |
| ** | целые, действительные, двойной точности, комплексные, комплексные двойной точности |
| = ~= | целые, действительные, двойной точности, комплексные, комплексные двойной точности, булевские, символьные |
| < <= > => | целые, действительные, двойной точности, символьные |
| ~ & \| | булевские |
| \|\| | скалярные, массивы |

Оператор возведения в степень является неопределенным, если основание отрицательно и экспонента не целое число, поскольку в этом случае результатом будет комплексное значение. Компилятор сгенерирует семантическую ошибку, если попытаться применить оператор к операнду или паре операндов, для которых он не определен.

Когда операнды имеют различные типы, один из них или оба могут быть приведены к одинаковому типу в соответствии с иерархией типов:




Данная иерархия является только частичным порядком. Если операнды имеют следующие типы: целый и двойной точности, то тип целый будет приведен к типу двойной точности, и результатом будет значение двойной точности; но если операторы имеют типы двойной точности и комплексный, то оба они будут приведены к типу комплексные двойной точности, того же типа будет и результат. Приведение типов распространяется и на вектора. Если нужно умножить вектор целых на вектор действительных, все элементы целочисленного вектора приводятся к типу действительный, и затем вектора перемножаются поэлементно. Результатом будет вектор действительных значений. Следующие выражения иллюстрируют приведение типов:

2 * 2.d0 = 2.d0 * 2.d0
3.5 mod 3 = 3.5 mod 3.0
2 * complex[1, 0] = complex[2.0, 0.0] * complex[1.0, 0.0]
2.0d0 * complex[1, 0] = double_complex[2.d0, 0.d0] * double_complex[1.d0, 0.d0]

### 1.6. Ошибочные значения

Множество значений для каждого типа данных (встроенного или определенного пользователем) дополнено одним ошибочным значением. Такое значение вырабатывается, если операция не может вернуть правильное значение.

Ошибочное значение возвращается, если:

1. Имеет место выход за границы диапазона типа данных.

2. Операция, конструирующая агрегатную структуру данных, неожиданно прерывает работу.
3. Входные данные операции не допустимы.
4. Производится попытка доступа к массиву за границами диапазона индексов.

Язык позволяет явно генерировать и тестировать на ошибку значения с помощью встроенных операций, error[typename] и is error[expression], соответственно. Следующее выражение дает пример применения этих операций:

```
let
  BadValue  := error[real];        % ошибка действительного типа
  FooError  := error[foo];         % ошибка пользовательского типа foo
  GoodValue := 3;
in
  is error[BadValue],              % True
  is error[GoodValue],             % False
  GoodValue = error[integer],      % error[boolean]
  BadValue = error[real],          % error[boolean]
end let
```

## 2. ПРОСТЫЕ ВЫРАЖЕНИЯ

### 2.1. Выражение let

*Выражение let* определяет множество имен и использует их для вычисления одного или более выражений. Каждое *выражение let* определяет свою область. Его форма приведена ниже:

```
let
    список определений
in
    список выражений
end let
```

*Список выражений* имеет арность и тип, значением выражения *let* является значение этого списка. Областью действия каждого имени, определенного в выражении, является внутренность выражения за исключением вложенных областей, которые могут переопределять данное имя. Имена, неопределенные в *let выражении*, импортируются из окружающей области. Рассмотрим пример:

```
let
    average  := (a + b + c) / 3.0;
    diff_a   := (a — average) ** 2;
    diff_b   := (b — average) ** 2;
    diff_c   := (c — average) ** 2
in
    average, (diff_a + diff_b + diff_c) / 2.0
end let
```

Это выражение определяет четыре имени: average, diff_a, diff_b и diff_c. Далее эти имена используются для вычисления двух значений: среднего арифметического и дисперсии чисел a, b и c.

Вложенные выражения могут локально переопределять имена, Новое значение скрывает все внешние определения. Например, в выражении

```
let
    a := 1;
    b := let a := 2.0 in 3.0 * a end let;
    c := integer(b)
in
    a + c    % 1 + 6
end let
```

определение b локально переопределяет a. Во внутренней области значением a является 2, и потому значение b есть 6.0. Переопределение никак не отразилось на значении a во внешней области действия, поэтому значением *let выражения* является 1 + 6.

### 2.2. Выражение if

Общая форма *выражения if* следующая:

```
if   булевское выражение then
    список выражений
else
    список выражений
end if
```

*Выражение if* может иметь и более двух ветвей; каждая добавочная ветвь имеет указанную ниже форму:

elseif *булевское выражение* then *список выражений*

Далее приведен пример *выражения if:*

```
person_type := if age < 5 then
        “infant”
      elseif age < 18 then
        “child”
      elseif age < 65 then
        “adult”
      else
        “senior”
      end if;
```

*Ветвь else* является обязательной, арность и тип каждой ветви должны быть идентичными.

Условия вычисляются в текстовом порядке, и список выражений, следующий за первым истинным условием, выполняется. Если какое-нибудь условие вычисляет ошибку до того, как встретится истинное условие, тогда все выражение возвращает ошибочные значения того же типа, что и тип ветвей.

### 2.3. Выражение case

Концептуально *выражение case* аналогично *выражению if*, но их синтаксис и семантика различны. Общая форма выражения case приведена ниже.

```
case управляющее выражение
      of список условий then список выражений
      of список условий then список выражений
  . . .
      else список выражений
end case
```

*Case* содержит управляющее выражение и одну или более условных ветвей. Управляющее выражение может выбирать по значению, тэгу союза или по сигнатуре типа. Арности ветвей должны быть идентичными, типы ветвей тоже должны совпадать за исключением случая выбора по сигнатуре типа (см. в главе 3.5 описание *выражения case type*).

Ниже приведен пример выражения case:

```
type_of_home := case pet
                of “fish”                    then “bowl”
                of “cat”, “dog”              then “house”
                of “mouse”, “hamster”, “bird” then “cage”
```

```
        else                    “n/a”
            end case
```

Значение управляющего выражения pet сравнивается с тестом каждой условной ветви в текстовом порядке до совпадения. Значение найденной ветви и будет значением всего *выражения case*. Каждая ветвь может иметь более одного тестирующего значения, например, вторая и третья ветвь в предыдущем коде. *Ветвь else* покрывает все значения, не включенные в тесты всех предыдущих ветвей.

В отличие от *выражения if*, *ветвь else* не обязательна. Обычно тесты *выражения case* покрывают все множество значений контрольного выражения, поэтому *ветвь else* не всегда нужна. В случае, когда *выражение case* не содержит *ветви else* и не найдено значение, совпадающее со значением управляющего выражения, выражение возвращает ошибочные значения тех же типов, что и ветви. Также *выражение case* возвращает ошибочные значения, если управляющее выражение вычисляет ошибку или ошибочное значение теста встречается раньше значения теста, совпадающего со значением управляющего выражения.

Если тип управляющего выражения допускает операции <= и >=, тогда в качестве теста можно использовать диапазон значений. Диапазон специфицируется следующим образом:

*нижняя_граница : верхняя_граница*

и содержит внутренние значения. Пример *выражения case*, использующего диапазоны, приведен ниже:

```
first_roll :=
  case die_1 + die_2
    of    2:3, 12        then “lose”
    of    7, 11          then “win”
    of    4:6, 8:10      then “no decision”
  end case
```

### 2.4. Выражение where

*Выражение where* является параллельным по данным условным выражением. Оно имеет следующий вид:

```
where выражение с булевским массивом  then
        список выражений
else
```

*список выражений*
end where

*Выражение where* может иметь более двух ветвей, каждая дополнительная ветвь имеет форму:

elsewhere *выражение с булевским массивом* then *список выражений*

Тест должен представлять собой функцию только от одного массива, на который ссылаются как на управляющий массив, а арность и тип ветвей должны быть идентичны.

Операционная семантика *выражения where* такова, что для каждого скалярного элемента управляющего массива вычисляется эквивалентное *выражение if* и результаты этих выражений собираются в массив, аналогичный по размерам и форме управляющему массиву. Рассмотрим пример

```
A_threshold := where    A < 0.0        then   -1.0
                 elsewhere  A = 0.0    then   0.0
                 else                         1.0
               end where
```

и предположим, что A есть двумерный массив $n \times m$, тогда для каждого из nm скалярных элементов A будет вычисляться *выражение if* вида

```
if A[i,j] < 0.0          then   –1.0
elseif A[i,j] = 0.0      then   0.0
else                            1.0
end if
```

и nm результирующих значений соберутся в массив таких же размера и формы, с той же нижней границей, что и A.

Если управляющий массив имеет ошибочное значение, тогда *выражение where* возвращает ошибочное значение того же типа, что и тип управляющего массива. Если же *выражение if* для конкретного элемента возвращает ошибочное значение, то ошибочное значение того же типа, что и тип этого элемента, будет находиться на месте этого элемента в результирующем массиве.

## 3. ДЕКЛАРАЦИИ ТИПОВ ДАННЫХ

Компилятор языка Sisal 90 может определить типы большинства имен из контекста их определения. Пользователю нет необходимости декларировать тип имени, как это сделано в языках Fortran и C, однако он может по желанию явно указывать его, например:

```
let
    pi:real := 3.14159;
    r:integer := 2;
in
    pi * r**2
end let
```

Явно указывать типы обязательно только при спецификации формальных параметров и результатов функции. Пользователь обязан декларировать тип каждого имени и результата в заголовке функции.

```
function f(a,b:integer; c:real returns integer, real)
```

Хотя такое явное типизирование не является необходимым с теоретической точки зрения, Sisal 90 требует этого для улучшения понимаемости программ. Декларирование типов программных входов в заголовке функции main помогает компилятору разрешать двусмысленности типов в исходном коде, устраняя необходимость дорогостоящих проверок во время исполнения.

Sisal 90 предопределяет восемь скалярных типов (boolean, character, integer, real, double, complex, double_complex, null) и пять типов конструкторов (array, stream, record, union, function). Для определения новых типов или для построения множества типов используется предложение декларации типа:

type *идентификатор типа = спецификация типа*

Спецификация типа может ссылаться на любой встроенный или определенный пользователем тип, но рекурсивные определения возможны только для союзов (union) и множества типов (type set). Внутри единицы компиляции каждый идентификатор типа должен быть уникален и продекларирован до своего использования. Типы, значения и функции имеют различные именные области, поэтому можно использовать одинаковые имена для типа, значения и функции.

### 3.1. Скалярные типы

Sisal 90 предопределяет восемь скалярных типов:

| | | | |
|---|---|---|---|
| integer | real | double | boolean |
| null | character | complex | double_comlex |

Для лучшего понимания и во избежание тонких ошибок использования типов можно переименовывать типы. Например, для различения градусной и радианной мер угла продекларируем следующие типы:

```
type Degree = double;
type Radian = double;
```

Переименованные типы могут оперировать только со своими базовыми типами, но не с другими переименованными типами или базовыми типами. Можно сложить значения типов Degree и double (результат будет иметь тип Degree), однако нельзя складывать значения типов Degree и Radian, Degree и integer, Degree и real, Degree и complex и т.д. Более того, нужно аккуратно использовать переименованные типы при вызове функции. Следующий пример показывает правильное и неправильное использование переименованных типов.

```
function   ops_on_angles(r:Radian; d:Degree
           returns Radian, Radian, Degree, Degree)
   r*2.0d,      % тип Double можно привести к типу Radian
   d+r,         % неправильно, градусы не являются радианами
   r/2.0,       % неправильно, тип real нельзя привести к типу
                % Radian
   d-180,       % неправильно, тип integer нельзя привести к
                % типу Degree
end function
```

Каждый определенный пользователем тип, переименовывающий другой тип, создает функцию преобразования с таким же именем. Функция преобразует объекты базового типа в объекты именованного типа, например:

```
x := Radian(2.0d0)
```

Следует отметить, что функция преобразования не конвертирует значения одного переименованного типа в другой, например Degree в Radian. Такая

конвертация обычно требует некоторой математической операции, а не просто изменения типа, что и демонстрирует следующая функция:

```
function degree_to_radian(d:Degree returns Radian)
    3.1415926535d0 / 180.0d0 * d
end function  % Degree_to_Radian
```

### 3.2. Агрегатные типы данных

Sisal 90 поддерживает четыре агрегатных типа данных:

array stream record union

**Массивы.** Массивы являются однородными агрегатными структурами, которые поддерживают прямой доступ. Каждый элемент этой структуры имеет индексное значение, которое уникальным образом идентифицирует его. Индексные значения непрерывны и по умолчанию имеют нижней границей единицу. Все массивы должны обязательно декларироваться. В главе 4.1 содержатся описания операций над массивами. Ниже приведены примеры деклараций типа массив.

```
type OneDim  = array[real];
type TwoDim  = array[OneDim];
type Strange = array[stream[OneDim]];
```

Тип OneDim определяет массив целых, это одномерный массив с базовым типом integer. Тип TwoDim определяет массив массивов целых, это двумерный массив с базовым типом integer. Наконец, тип Strange определяет массив записей типа OneDim, это одномерный массив с базовым типом stream[Onedim].

Можно определить массив с любым встроенным или определенным пользователем базовым типом, но рекурсивные определения недопустимы. Декларация типа массив не специфицирует нижнюю границу, размер и форму. Такие атрибуты являются следствием значения массива, а не его типа. Массивы разных типов могут быть операндами одной операции, если значения их базовых типов могут быть операндами этой операции. С другой стороны, массив несоответствующего типа не может быть послан в качестве фактического параметра при вызове функции.

**Потоки.** Как и массивы, потоки являются однородными структурами, но они не поддерживают прямого доступа. Элементы потока доступны в

последовательном порядке. Прежде чем использовать второй элемент, необходимо использовать первый. В главе 4.2 описаны операции над потоками. Ниже приведены декларации типа поток:

```
type StrCmplx = stream[complex];
type StrArray = stream[OneDim];
```

Тип StrCmplx определяет поток комплексных чисел, это одномерный поток с базовым типом complex. Тип StrArray определяет поток массивов целых, это поток с базовым типом OneDim.

Можно определить поток с любым встроенным или определенным пользователем базовым типом, но рекурсивное определение потока запрещено. Размер и форма потока являются следствием его значения, а не декларации типа. Потоки различных типов могут быть операндами одной операции, если значения их базовых типов могут быть операндами этой операции, однако поток несоответствующего типа не может быть послан в качестве фактического параметра при вызове функции. Потоки могут быть реализованы как нестрогие структуры данных. Такая реализация делает элементы потока доступными до полного определения самого потока.

**Записи.** Запись — это возможно разнородная структура данных, состоящая из одного или более полей значений. Два следующих примера представляют встроенные типы complex и double_complex.

```
type complex = record[real: real; imag: real];
type double_complex = record[real, imag: double_complex];
```

Порядок полей является существенным, так тип

```
type my_complex = record[imag, real: real];
```

не эквивалентен типу complex.

Каждое поле имеет уникальное в декларации имя и тип. Типы полей могут совпадать или различаться. Любой встроенный или определенный пользователем тип может выступать в качестве типа поля. Рекурсивное определение записи недопустимо. В качестве типа поля может выступать как имя типа, так и спецификация типа, например, array[character]. В главе 4.3 описаны операции над записями.

**Союзы.** Союз является разнородной структурой данных, чьи компоненты могут иметь различные типы. Декларация союза представляет собой множество из одного или более определенных пользователем имен тэгов. Каждое имя тэга должно быть уникальным, тип представляется идентифи-

катором или спецификацией (по умолчанию типом тега является null). Возможно рекурсивное определение союза. Несколько тэгов могут иметь одинаковые типы. В главе 4.4 описаны операции над союзами.

Бинарное дерево является хорошим примером типа союз. Это иерархическая структура данных, содержащая поддеревья и листья. Вершинами дерева будут либо корни поддеревьев, либо листья. Ниже приведена декларация типа для бинарного дерева, в листьях которого хранятся действительные значения.

```
type tree =     union[root:record[left,ridth:tree];
      leaf:real];
```

Другим использованием союзов является генерация перечислимых типов. Следующие декларации типов

```
type rgb = union[red, green, blue];
type hsv = union[cyan, yellow, magenta];
```

определяют тип для цветов красный, зеленый и синий, а второй тип для цветов голубой, желтый, пурпурный. По умолчанию тип каждой компоненты есть null, а значение — nil.

### 3.3. Функции

Функции Sisal 90 наряду с типами являются первостепенными объектами, это означает, что они могут выступать в качестве параметров и результатов других функций. Ниже приведены декларации типа функция.

```
type OneParm = function[real returns real];
type TwoParm = function[real, real returns real];
```

Функция типа OneParm работает с одним действительным значением и возвращает одно действительное значение, функция типа TwoParm работает с двумя действительными значениями и возвращает одно действительное значение.

Функциональные типы действуют, как и все другие типы. Идентификаторы функциональных типов могут стоять там же, где и идентификаторы типов: после имен

```
F:OneParm := f(_,3.0);
```

или в заголовке функции

```
function integrate(n:integer; a,b:real; f:OneParm returns real)
```

### 3.4. Множество типов

*Множество типов* — это множество разделенных запятой встроенных или определенных пользователем типов. Ниже приведены примеры множеств общих математических типов.

```
type floats = [real, double];
type non_complex = [integer, floats];
type all_complex = [complex, double_complex];
type numeric = [all_complex, non_complex];
```

Имя типа floats имеет либо действительное, либо двойной точности значение, имя типа numeric — это значение одного из следующих типов: integer, real, double, complex, double_complex. Sisal 90 предопределяет множество типов any, которое покрывает все типы.

Множества типов обеспечивают некоторую степень полиморфизма. Например, можно определить единственную функцию для нахождения наибольшего значения в массиве целых, действительных или двойной точности значений.

```
function largest_value(A:array[non_complex] returns non_complex)
    for element in A
    returns greatest of element
    end for
end function  % largestt_value
```

Множества типов могут определяться рекурсивно. Например, декларация

```
type integer_or_array = [integer, array[integer_or_array]]
```

определяет множество типов, которое включает целые и все n-мерные массивы целых, где $n \geq 1$. Экземпляр типа integer_or_array есть либо целое, либо массив целых, либо массив массивов целых и т.д.

Хотя множество типов похоже на союз, эти типы различны. Первый является инструментом для поддержания полиморфизма, второй определяет разнородную структуру данных.

### 3.5. Выражение case type

*Выражение case type* аналогично *выражению case*, только выбор ветви происходит по типу. Ветви *выражения case type* должны совпадать по арности, но могут иметь разный тип. Если тип управляющего параметра известен во время компиляции, компилятор может подставить вместо выра-

жения соответствующую ветвь. Следующая функция показывает пример использования *выражения case type*.

```
function zero(x:numeric returns numeric)
  case type x
    of integer          then 0
    of real             then 0.0
    of double           then 0.0d0
    of complex          then complex(0,0)
    of double_complex   then double_complex(0,0)
  end case
end function
```

Тип возвращаемого нуля зависит от типа значения x.

Тип, который стоит в тестах, может быть встроенным типом, агрегатным типом (array, stream, record, union), определенным пользователем типом (включая тип функция и множество типов), а также проверкой типа. Имя агрегатного типа идентифицирует любой агрегат этого типа вне зависимости от формы, размера и типа компонент. Например, выражение

```
case type A
    of integer, real, double   then A
    of array, stream           then A[1]
end case
```

возвращает A, если A имеет тип целый, действительный или двойной точности, или первый элемент A, если A является массивом или потоком. Заметим, что если тип A не совпадает ни с одним из перечисленных типов, то компилятор, если он знает тип A, или система поддержки исполнения сгенерирует ошибку несоответствия типов.

Наконец, проверка типа поддерживает выбор ветви на основе совпадения типов управляющего и тестового выражений. Например,

```
function  type_check(A,B,C:any returns array[character])
  case type A
    of type(B) then “same type as B”
    of type(C) then “same type as C”
    else            “type mismatch”
  end case
end function
```

## 4. АГРЕГАТНЫЕ ВЫРАЖЕНИЯ

Sisal 90 поддерживает четыре типа агрегатных структур: массивы, потоки, записи и союзы. Массивы и потоки являются однородными структурами, причем их размер и форма есть характеристики времени исполнения. Записи и союзы являются разнородными структурами с фиксированным числом типов полей или тэгов. Поскольку определение союза может быть рекурсивным, экземпляр типа союз (например, дерево или лист) может быть произвольно большим во время исполнения. Эта глава содержит описание различных операций для создания и манипулирования объектами агрегатных данных.

### 4.1. Массивы

Массивы являются однородными структурами данных, которые поддерживают прямой доступ к элементам. Они имеют нижнюю границу индексов и содержат ноль или более элементов. Так как размер и форма массива — характеристики динамические, Sisal 90 содержит три предопределенные функции для определения размера массива, нижней и верхней границы его индексов:

```
liml(A)    % нижняя граница массива A
limh(A)    % верхняя граница массива A
size(A)    % число элементов в массиве A
```

#### *4.1.1. Создание массивов*

Массивы языка Sisal 90 подобны массивам в традиционных языках программирования. Однако, поскольку Sisal 90 — язык функциональный, массивы в нем создаются за одну операцию, а не последовательностью операторов присваивания.

Прямое указание нижней границы и перечисление элементов массива является простейшим путем его построения.

```
A := array[1!6,0,2,2,5,2];      % нижняя граница 1
B := array[5!5,5,5,1,2,1,2];    % нижняя граница 5
C := array[-5!5,5,5,1,2,1,2]    % нижняя граница –5
```

Слово array и/или нижнюю границу можно опустить. По умолчанию нижней границей считается единица.

```
A := [6,0,2,2,5,2]     % эквивалентное определение для массива A
```

Для удобства язык содержит предопределенную функцию fill(l,h,v), которая строит массив с нижней границей l, верхней границей h и с (h-l+1) элементами, значением которых является v.

```
fill(1,5,0)   % array[1!0,0,0,0,0]
```

Хотя компилятор в состоянии определить тип массива, пользователь по желанию может включить имя типа в определение массива. Допустим, что OneDim определен следующим образом:

```
type OneDim = array[integer]
```

Тогда определение A можно записать в виде

```
array OneDim [6,0,2,2,5,2];
```

Более того, в языке существует конструктор array и конструкторы для каждого продекларированного типа массива. Конструктор работает только со списком значений элементов и всегда возвращает массив с нижней границей 1. Эквивалентные определения массива A с использованием конструктора приведены ниже:

```
A := array{6,0,2,2,5,2};
A := OneDim(6,0,2,2,5,2}
```

В языке Sisal 90 существует ряд операций, которые создают новые массивы из уже существующих. Список этих операций приведен ниже:

```
D := A[1!5];        % замещение
E := A[1!5,6,7];    % замещение
F := A[2:4];        % выбор подмассива
G := A[2:6:2];      % выбор подмассива с шагом 2
H := reml(A);       % удаление элемента с нижним индексом
I := remh(A);       % удаление элемента с верхним индексом
J := addl{A,3};     % добавление элемента в начало
K := addh(A,3);     % добавление элемента в конец
L := setl(A,0);     % установка нижней границы
M := A||B           % конкатенация
```

Массив D эквивалентен массиву A, за исключением первого элемента, значением которого является 5. Определение массива E показывает, как с помощью одной операции можно заместить несколько элементов. В данном примере замещаются элементы 1, 2, 3. Ниже приведены два эквивалентных определения массива E:

```
E := A[1!5; 2!6; 3!7];
E := A[1!5] [2!6] [3!7]
```

Более подробную информацию можно получить в главе 4.1.3, где обсуждаются выбор подмассивов и индексные операции.

Определения массивов F и G иллюстрируют операцию *выбора подмассива*. Эти предложения определяют два новых массива с нижними границами 1, которые содержат второй, третий, четвертый и второй, четвертый, шестой элементы массива A соответственно.

Определения массивов H, I, J, K, L содержат вызов предопределенных функций, названия которых объясняют их работу. Каждая функция возвращает новый массив, содержащий некоторые элементы входного массива. Нижняя граница массивов, возвращаемых функциями reml и addl, соответственно, на единицу больше и на единицу меньше нижней границы входного массива. Последнее выражение иллюстрирует конкатенацию массивов. Операция возвращает один массив, собранный из элементов массива A, за которыми следуют элементы массива B. Если какой-либо из массивов пуст, то он не вносит никакого вклада в результирующий массив. Нижняя граница нового массива равна нижней границе массива A.

Все вышеперечисленные операции можно применять, не опасаясь эффекта копирования. Так же как для языка Sisal 1.2, компилятор языка Sisal 90 удалит ненужные операции копирования. Можно ожидать, что большинство операций над массивами будут выполняться в заранее отведенном для них месте. В качестве примера рассмотрим следующую функцию, которая работает с массивом `queue`, где могут быть удалены как первый, так и последний элементы.

### *4.1.2. Векторные операции над массивами*

Sisal 90 поддерживает векторные расширения для арифметических, булевских операторов и операторов отношений. Если A и B массивы или потоки любой размерности, тогда A*B обозначает покомпонентное произведение их элементов. Математические определения некоторых векторных операций языка для 2-D массивов приведены ниже:

$$A * B = [c_{ij} \mid c_{ij} = a_{ij} * b_{ij}]$$
$$A ** B = [c_{ij} \mid c_{ij} = a_{ij} ** b_{ij}]$$
$$A \bmod B = [c_{ij} \mid c_{ij} = a_{ij} \bmod b_{ij}]$$
$$A <= B = [c_{ij} \mid c_{ij} = (a_{ij} <= b_{ij})]$$
$$A \,\&\, B = [c_{ij} \mid c_{ij} = a_{ij} \,\&\, b_{ij}]$$

Заметим, что * не матричное умножение, а покомпонентное произведение элементов массива.

Базовый тип массива или потока должен соответствовать оператору. Базовым типом результата будет тип, сгенерированный оператором по типам операндов (преобразования типов описаны в главе 1.5). Например, для четвертого определения базовым типом для A и B могут служить integer, real, double, character, а типом результата будет boolean. Для последнего определения базовым типом может быть только boolean, и результирующим типом тоже будет boolean.

Покомпонентные операции рассматривают пары элементов массивов или потоков по каждой позиции и к каждой паре применяют операцию. Если элементами в свою очередь являются массивы или потоки, тогда операция применяется рекурсивно. Рассмотрим следующее *выражение let*:

```
let
  A := [1! 11,22,33];
  B := [2! 10,20,30]
in
  A*B    % [11*10,22*20,33*30]
end let
```

Если размеры массивов не совпадают, тогда более короткий вектор дополняется ошибочными значениями, как в следующем примере:

```
let
  A := [1! 11,22,33];
  B := [2! 10,20]
in
  A*B   % [11*10, 22*20, 33*error]
end let
```

Нижней границей результата всегда является 1, а размер результата всегда равен размеру большего массива. Установка 1 в качестве нижней границы результата не случайна, она необходима для сохранения коммутативности. Например:

```
let
  A := [1! 11,22,33];
  B := [2! 10,20,30]
in
  A*B,     % [1! 11*10,22*20,33*30]
```

```
  B*A,     % [1! 11*10,22*20,33*30]
end let
```

Если один операнд является скаляром, а другой массивом или потоком, то скаляр преобразуется в массив или поток того же размера и формы, что и другой операнд, далее операция применяется, как указывалось выше. Выражение

```
let
  B := [2! 10,20,30]
in
  2*B    % [2,2,2]*[2! 10,20,30]
end let
```

возвращает массив с нижней границей 1, того же размера, что и B. Наконец, если оба операнда являются потоками, то результатом тоже будет поток.

Как указывалось ранее, структурные операнды (массивы и потоки) могут иметь любое количество размерностей, в действительности размерности операндов не обязательно совпадают. Эта последняя черта отличает Sisal 90 от других языков, поддерживающих векторные операции. Векторная операция рекурсивно собирает в пары соответствующие элементы операндов до тех пор, пока операнды не станут скалярами, и затем применяет оператор. Результирующие значения собираются в структуру тех же формы и размера, что и операнд с большим числом размерностей.

Следующий пример прояснит семантику векторных операций. Пусть A есть двумерный массив

```
[[1,2,3],[4,5,6],[7,8,9]],
```

а B — трехмерный массив

```
[2! [[10,20,30],[40,50,60],[70,80,90]],
    [[11,21,31],[41,51,61],[71,81,91]],
    [[12,22,32],[42,52,62],[72,82,92]
]
```

Операция A*B выполняется следующим образом. Во-первых, элементы A и B покомпонентно собираются в пары

```
[1,2,3]*[[10,20,30],[40,50,60],[70,80,90]]
[4,5,6]*[[11,21,31],[41,51,61],[71,81,91]]
[7,8,9]*[[12,22,32],[42,52,62],[72,82,92]]
```

Поскольку каждая подоперация является произведением вектора на вектор, для них применяем рекурсию. Первые три подоперации будут такими:

1*[10,20,30], 2*[40,50,60], 3*[70,80,90]

Каждая операция теперь является умножением скаляра на вектор. Скалярное значение распространяется до массива таких размера и формы, что и парный ему элемент

[1,1,1]*[10,20,30]
[2,2,2]*[40,50,60]
[3,3,3]*[70,80,90]

и операция применяется снова. Теперь получилось девять обычных операций умножения скаляров:

1*10, 1*20, 1*30
2*40, 2*50, 2*60
3*70, 3*80, 3*90

Девять результатов собираются для создания массива тех же формы и размера, что и первоначальный операнд с большим числом размерностей. В данном случае операндами являются первый элемент массива A, одномерный массив, и первый элемент массива B, двумерный массив с тремя строками и тремя столбцами. Таким образом, результатом будет двумерный массив

[[10,20,30], [80,100,120], [210,240,270]]

Вторая и третья подоперации тоже вернут двумерные массивы в качестве результатов. Из всех промежуточных результатов соберется конечный результат данной векторной операции A*B

```
[[[10, 20, 30],   [ 80,100,120],   [210,240,270]]
 [[44, 84,124],  [205,255,305],  [426,486,546]]
 [[84,154,224], [336,416,496],  [648,738,828]]
]
```

### *4.1.3. Выборка подмассивов*

Sisal 90 поддерживает несколько способов ссылки на элементы или множества элементов массива для операции выборки. Среди них можно выделить синглеты (singlets), дуплеты (douplets), триплеты (triplets) и индексные массивы (index arrays).

**Синглеты.** Синглетом является целая константа или выражение единичной арности целого типа. Любой элемент массива можно выбрать с помощью его индекса — синглета. Если значение индекса не попадает в диа-

пазон, указанный границами массива, т.е. он слишком мал, слишком велик или является ошибочным значением, то результатом является ошибочное значение того же типа, что и элемент массива. Следующее *выражение let* иллюстрирует выборку по синглету:

```
let
    i := 5;
    A := array[2! 11,22,33,44,55,66]
in
    A[2],    % 11
    A[i],    % 44
    A[i-1],  % 33
    A[-3],   % error[integer] (индекс слишком мал)
    A[8]     % error[integer] (индекс слишком велик)
end let
```

Заметим, что если нижняя граница массива не единица, то индекс и позиция элемента в массиве не эквивалентны. Если нижней границей массива является ошибочное значение, к элементам массива нет прямого доступа.

Многомерные массивы в языке Sisal 90 концептуально рассматриваются как массивы массивов. M-мерный элемент n-мерного массива, где $m \leq n$, адресуется указанием внешних $n - m$ индексов массива. Например, если A является пятимерным массивом, то выражение A[2][3] указывает на третий элемент второго элемента A, который является трехмерным массивом. Индексы всегда рассматриваются слева направо, с самой верхней размерности до самой нижней. Многомерный индекс можно записывать в виде списка синглетов, например A[2,3].

В качестве многомерного индекса можно использовать любое целое выражение арности m. Пусть функция N_2d определена следующим образом:

```
function N_2d(n,k: integer returns integer, integer)
    n/k, n mod k
end function,
```

тогда выражение A[N_2d(10,5)] является правильным и возвращает элемент A, стоящий в строке 2, в столбце 0.

**Дуплеты и триплеты.** Новым для Sisal 90 механизмом выборки является выборка по триплетам. Триплеты используются для спецификации однообразно расположенных элементов массива. Триплет определяется тремя целыми числами, разделенными двоеточием:

*первый : второй : шаг*

Для удобства триплет с шагом 1, записывается в виде дуплета:

*первый : второй*

Первое и второе значения по умолчанию полагаются равными соответственно нижней и верхней границе массива. Шаг по умолчанию равен 1.

Следующее *выражение let* иллюстрирует триплетную нотацию:

```
let
    A := array[11.1, 22.2, 33.3, 44.4, 55.5]
in
    A[1:5],      % array[11.1, 22.2, 33.3, 44.4, 55.5]
    A[:],        % array[11.1, 22.2, 33.3, 44.4, 55.5]
    A[1:5:2],    % array[11.1, 33.3, 55.5]
    A[5:2:-2],   % array[55.5, 33.3, 11.1]
    A[2:5:2],    % array[22.2, 44.4]
    A[1:0],      % array[]
    A[1:1:0]     % error array
end let
```

Следует отметить, что результирующий массив всегда имеет нижнюю границу 1. Если *второе* меньше *первого* при положительном *шаге* или *первое* меньше *второго* при отрицательном *шаге*, тогда возвращается пустой массив с нижней границей 1. Если шаг 0, то возвращается ошибка типа массив. Следующая формула вычисляет число выбранных элементов для положительного и отрицательного шагов:

*число_выбранных_элементов(first, second, step) = max(0,(second — first + step)/step)*

Применяя формулу к вышестоящему примеру, получим:

*число_выбранных_элементов(1, 5, 1) = max(0,(5 — 1 + 1) / 1) = 5*
*число_выбранных_элементов(1, 5, 2) = max(0,(5 — 1 + 2) / 2) = 3*
*число_выбранных_элементов(5, 1, -2) = max(0,(1 — 5 - 2) / -2) = 3*
*число_выбранных_элементов(2, 5, 2) = max(0,(5 — 2 + 2) / 2) = 2*
*число_выбранных_элементов(1, 0, 1) = max(0,(0 — 1 + 1) / 1) = 0*
*число_выбранных_элементов(1, 1, 0) = max(0,(1 — 1 + 0) / 0) = не определено*

**Индексные массивы.** Sisal 90 поддерживает индексные массивы для выборки элементов из массива. Пусть V — массив

V := array[11.1, 22.2, 33.3, 44.4, 55.5],

и пусть Indices — массив целых

Indices := array[3, 3, 2, 2, 1, 1, 4, 4],

тогда выражение V[Indices] эквивалентно массиву

array[33.3, 33.3, 22.2, 22.2, 11.1, 11.1, 44.4, 44.4]

Построенный массив всегда имеет нижнюю границу 1 и содержит столько элементов, сколько индексов включает в себя индексный массив. Индексный массив может содержать больше или меньше значений, чем первоначальный массив. В различных научных приложениях часто используются индексные массивы. Присутствие ошибок усложняет семантику операции выборки по индексному массиву. Если исходный или индексный массив являются ошибкой типа массив или нижняя граница исходного массива ошибочна, тогда сконструированный массив имеет нижнюю границу 1, размер, равный размеру индексного массива, и содержит только ошибочные значения. Если ошибкой является нижняя граница индексного массива, на конструирование это никак не повлияет, так как данная операция не является функцией от нижней границы массива индексов. Если исходный массив содержит ошибочный элемент, то в конструируемом массиве при выборе этого элемента стоит ошибочное значение. Если индексный массив содержит ошибочный элемент, соответствующий этому индексу элемент конструируемого массива тоже будет ошибочным.

### *4.1.4. Операция многомерной выборки подмассивов*

Sisal 90 позволяет, используя синглеты, дуплеты, триплеты и индексные массивы, выбирать m-мерные массивы из n-мерных массивов, где $m \leq n$. Для осуществления такой выборки имеется два оператора — cross и dot, при этом важно учитывать размерность выборки и размерность выбираемого элемента. Размерность результирующего массива будет равна сумме этих двух размерностей. Если проводится двумерная выборка и каждый выбираемый элемент имеет размерность 3, то результирующим будет 5-мерный массив. Простейшей многомерной выборкой подмассива является серия cross-произведений дуплетов или триплетов. Пусть A — это массив

```
array[  array[11,12,13,14,15,16,17],
        array[21,22,23,24,25,26,27],
     array[31,32,33,34,35,36,37]],
```

тогда выражение A[1:3 cross 1:3] или A[1:3, 1:3][1] возвращает массив

array[array[11,12,13], array[21,22,23], array[31,32,33]],

а выражение A[1:3:2, 1:7:2]возвращает массив

[1] Запятая является синонимом слова `cross`.

```
array[ array[11,13,15,17], array[31,33,35,37]]
```

В обоих примерах выборка имеет размерность 2, а выбираемые элементы 0-мерны, таким образом, результатом будет двумерный массив. Как всегда нижняя граница как самого массива, так и компонент массива равна 1. Оператор cross собирает в пары каждый элемент своего левого оператора с каждым элементом своего правого оператора. Выбранные элементы исходного массива собираются в результирующий массив построчно.

В качестве обоих операндов операций dot и cross можно использовать индексные массивы. Например, выражение

```
let
    V := array[1,2,3,4,5,7]
in
    A[1:2 cross V]
end let
```

возвращает массив

```
array[ array[11,12,14,15,17], array[21,22,24,25,27] ]
```

Если операндом cross оператора является синглет, то его элементы образуют пары с каждым элементом другого операнда, элементы исходного массива выбираются, и результирующий массив собирается построчно, синглет не увеличивает размерность выборки. Поэтому cross произведение несинглета на синглет выбирает одномерный массив, а cross произведение синглета на синглет выбирает 0-мерный массив или просто значение. Это правило важно, так как A[1 cross 1] выбирает элемент массива A по строке 1 и столбцу 1, однако чтобы выбрать третий столбец A, одномерный объект, необходимо использовать выражение A[ :, 3].

Оператор dot собирает в пары каждый элемент первого операнда с соответствующим элементом второго операнда, таким образом [1:3 dot 1:3] есть множество индексов {[1,1], [2,2], [3,3]}. Если операнды имеют различную длину, более короткий дополняется ошибочными значениями. Оператор dot полезен для выборки диагональных элементов массива. Например, пусть B есть массив

```
array[ array[11,12,13], array[21,22,23], array[31,32,33] ],
```

тогда выражения B[1:3 dot 1:3] и B[3:1:-1, 1:3] возвращают соответственно массивы array[11,22,33] и array[31,22,13]. Оператор dot в качестве операндов воспринимает только дуплеты, триплеты и индексные массивы, и выборка имеет размерность 1. Если попытаться применить оператор dot к синглету, компилятор выдаст синтаксическую ошибку.

Можно описать даже более сложную выборку, давая имена элементам дуплета, триплета или индексного массива, и затем используя эти имена в индексных выражениях. Рассмотрим выражение

B[ i in 1:3, 1:i]

Запятая, или cross произведение, образует пары из каждого элемента первого диапазона с каждым элементом второго диапазона. Имя i принимает значения из первого диапазона по очереди и затем используется для задания верхней границы второго диапазона. Индексное выражение генерирует множество индексов {[1,1], [2,1], [2,2], [3,1], [3,2], [3,3]}, и исходное выражение возвращает нижний треугольный подмассив массива B.

array[ array[11] array[21,22] array[31,32,33] ]

Одна операция выборки может включать произвольное число операторов dot и cross. Серии операций dot и cross вычисляются слева направо. Размерность выборки равна сумме дуплетов, триплетов и индексных векторов за вычетом числа операторов dot. Например, пусть C — это массив

```
array[  array[ array[111,112], array[121,122] ],
        array[ array[211,212], array[221,222] ],
        array[ array[311,312], array[321,322] ] ],
```

тогда выборка C[1:2 dot 1:2 cross 1] имеет размерность 1 (два дуплета минус один оператор dot) и генерирует множество индексов {[1,1,1], [2,2,1]}. Результатом будет массив array[111,221]. Следующий список дает примеры цепочек dot-cross произведений, их размерности и множества индексов, которые они генерируют (подразумевается, что U и V являются векторами целых с нижней границей 1, которые содержат по три элемента).

| dot-cross произведение | Размерность | множество индексов |
|---|---|---|
| 1, 1, 1:3 | 1 | [1,1,1], [1,1,2], [1,1,3] |
| U dot V | 1 | [U1, V1], [U2,V2], [U3, V3] |
| 1:3 dot U cross V | 2 | [1,U1,V1], [1,U1,V2], [1,U1,V3], [2,U2,V1], [2,U2,V2], [2,U2,V3], [3,U3,V1], [3,U3,V2], [3,U3,V3] |
| 1:3 cross 3:5 dot 1:5:2 | 2 | [1,3,1], [1,4,3], [1,5,5], [2,3,1], [2,4,3], [2,5,5], [3,3,1], [3,4,3], [3,5,5] |

### *4.1.5. Замещение подмассивов*

Sisal 90 позволяет замещать любое множество элементов, которое можно указать с помощью операций выборки, сохраняя при этом семантику однократного присваивания. Ниже приведена общая форма этой операции:

*имя_массива[выражение выбора! выражение замещения]*

Выражение создает новый массив, идентичный массиву *имя_массива*, за исключением замещенных элементов.

Любое множество элементов, выбранное описанными ранее методами, может быть замещено. Возможно заместить единственный элемент, непрерывное или разрывное множества элементов, диагонали и произвольное множество элементов, используя индексные массивы. Если *выражение выбора* является синглетом или разделенный запятыми список синглетов, тогда *выражение замещения* должно быть единственным значением или разделенным запятыми списком значений того же типа, что и выбранные элементы. Список значений замещает элементы слева направо, начиная с первого индекса. Следующее выражение иллюстрирует синглетное замещение в четырехмерном массиве A:

```
let
    B := array[array[array[1!2,3,4]]];   % трехмерный массив
    C := array[array[1!2,3,4]]           % двумерный массив
in
    A[1!B]                               % A[1] = B
    A[1!B,B]                             % A[1] = B, A[2] = B
    A[1,1!C,C]                           % A[1,1] = C, A[1,2] = C
end let
```

Если *выражение выбора* не является синглетом, тогда оно содержит как минимум один дуплет, триплет, индексный массив или цепочку dot-cross произведений. Пусть *имя_массива* обозначает n-мерный массив с базовым типом *x*. Тогда *выражение выбора* определяет m-мерный массив p-мерных массивов с базовым типом *x*, где

1) $0 \le p \le n-1$,
2) m — это сумма дуплетов, триплетов и индексных векторов минус количество dot операторов в *выражении выбора*,
3) p — это n минус сумма синглетов, дуплетов, триплетов и индексных массивов в *выражении выбора*.

*Выражение замещения* — это либо

- m-мерный массив p-мерных массивов с базовым типом *x* — оператор замещения применяется покомпонентно и рекурсивно m раз — либо
- p-мерный массив с базовым типом *x* — в массиве замещается каждый выбранный массив.

Пусть A — четырехмерный массив целых. Тогда

```
A[1:n, 1:n ! …], m=2, p=2;
A[1, 1:n ! …], m=1, p=2;
A[1, 1:n dot 1:n ! …], m=1, p=1;
A[1:n dot 1:n dot 1:n dot 1:n ! …], m=1, p=0
```

В качестве примера покомпонентного, рекурсивного применения оператора замещения рассмотрим результат выражения A[2:6, 2:4 ! B], где A — массив

```
array[ array[11,12,13,14,15]
       array[21,22,23,24,25]
       array[31,32,33,34,35]
       array[41,42,43,44,45]
       array[51,52,53,54,55]
       array[61,62,63,64,65]
       array[71,72,73,74,75]
     ]
```

и B — массив

```
array[ array[110]
       array[210,220]
       array[310,320,330]
       array[410,420,430,440]
       array[510,520,530,540,550]
     ]
```

Выражение замещения является правильным и возвращает массив

```
array[ array[11, 12,  13,  14,15]
       array[21,110,error,error,25]
       array[31,210, 220,error,35]
       array[41,310, 320, 330,45]
       array[51,410, 420, 430,55]
       array[61,510, 520, 530,65]
       array[71, 72,  73,  74,75]
     ]
```

В этом случае m = 2, p = 0. Оператор замещения собирает в пары второй элемент A и первый элемент B, третий элемент A и второй элемент B и т.д. Затем оператор применяется снова к каждой паре m раз. Так как первый и второй элементы B имеют один и два элемента соответственно, вместо отсутствующих элементов подставляются ошибочные значения. Остальные значения в элементах 4 и 5 массива B игнорируются.

Наконец, выражение

```
A[ 1, 1:5 ! 0;   % первая строка
   2:6, 1 ! 0;   % первый столбец
   2:6, 5 ! 0;   % последний столбец
   7, 1:5 ! 0    % последняя строка
 ]
```

возвращает массив, идентичный A, только окаймленный нулями.

## 4.2. Потоки

Потоки являются однородными агрегатными структурами данных. Различие между массивами и потоками в том, что первые поддерживают прямой доступ, а вторые — последовательный. Многие важные приложение, включая большинство программ обработки сигналов, работают с потоками данных. Приложения используют элементы индивидуально и по порядку. Так как потоки введены в первую очередь для поддержки таких приложений, множество операций над ними содержит только операции построения, доступа и удаление первого элемента и целого потока.

### *4.2.1. Построение потока*

Простейшим способом построить поток является перечисление множества его значений.

```
A := stream[6,0,2,2,5,2];   % поток целых
```

Ключевое слово stream обязательно, если его опустить, то будет построен массив с нижней границей 1. Хотя компилятор всегда может определить тип потока, при желании можно явно указывать имя типа потока. Если OneStr определить как

```
type OneStr = stream[integer];
```

тогда определение A можно записать в виде

```
A := stream OneStr [6,0,2,2,5,2];
```

Более того, можно использовать генерирующий конструктор stream или типовый конструктор, неявно определенный для каждого типа потока. Эквивалентным определением A, использующим типовый конструктор, является

```
A := stream{6,0,2,2,5,2};
```

и

```
A := OneStr{6,0,2,2,5,2};
```

В языке Sisal 90 существует несколько предопределенных функций, работающих с потоками. Все они перечислены ниже.

```
B := size(A);      % размер потока
C := empty(A);     % проверка потока на пустоту
D := first(A);     % удаление первого элемента потока
E := rest(A);      % поток без первого элемента
F := A || B        % конкатенация
```

Функция size возвращает число элементов в потоке. Так как поток — структура динамическая, то эта функция не может вернуть результат до полного построения потока.

Булевская функция empty возвращает false , если поток содержит один или более элементов, и true, если поток пуст и его построение завершено. Если конструирование еще не завершилось, функция ждет, пока либо построится элемент, либо закончится построение потока. Только пустой поток имеет нулевую длину.

Выражения для D и E возвращают, соответственно, первый элемент A и поток элементов A без первого элемента. Если A — пустой поток, то функции first и rest возвращают, соответственно, error[integer] и error[OneStr]. Чтобы прочитать i-й элемент потока, необходимо составить выражение

```
first( rest( rest . . .rest(A) . . . ))
```

Выражение содержит i-1 вызов функции rest. Так как это выражение слишком длинное и трудно читается, язык Sisal 90 поддерживает и более короткую нотацию A[i]. В отличие от элементов массива i-й элемент потока не доступен, пока первые i-1 элементов не произведены и не использованы.

Последнее выражение иллюстрирует конкатенацию потоков. Операция возвращает единый поток, составленный из элементов потока A, за которыми следуют элементы B. Если один из потоков пуст, то он не вносит вклад в содержание результирующего потока F.

### *4.2.2. Векторные операции над потоками*

Sisal 90 поддерживает расширения арифметических, булевских операций и операций отношения над типом данных поток. Семантика этих операций идентична семантике векторных операций над массивами и детально описана в главе 4.1.2.

### *4.2.3. Применение параллелизма "поставщик-потребитель"*

Главной целью использования потоков (подразумевается нестрогая реализация) является внедрение параллелизма "поставщик-потребитель" (producer-consumer) и динамических цепочек задач. Рассмотрим простейший пример — Решето Эратосфена. Алгоритм работает с целым аргументом n и возвращает простые числа, меньшие либо равные n. Логически алгоритм состоит из линейной последовательности "решет". I-ое решето работает над потоком целых, удаляет из потока числа, кратные i-ому простому числу, и возвращает как простое число, так и модифицированный поток целых. Заметим, что первый элемент во входном потоке есть i-ое простое число, так как оно является самым маленьким числом, которое не делят предыдущие простые числа. Параллелизм достигается, если все решета работают одновременно. Следующий код реализует функцию "решето":

```
function sieve(x: OneStr returns integer, OneStr)
  first(x),
    for element in rest(x) returns stream of
      element when element mod first(x)~=0
  end for
end function          % sieve
```

Приведенная ниже функция инициализирует вычисления, динамически создает и связывает вместе решета:

```
function Eratosthenes(n: integer returns stream[integer])
    for initial
      prime := 1;
      x     := for i in 2, n returns stream of i end for
      while ~empty(x) repeat
      prime, x := sieve(old x)
    returns stream of prime
    end for
end function  % Eratosthenes
```

Каждый вызов функции sieve в качестве входа берет поток, сгенерированный предыдущим вызовом, и свой выходной поток подает на вход следующему вызову. Если потоки реализованы нестрого, тогда *выражение for initial* динамически конструирует линейную цепочку задач, выполняющуюся одновременно. Алгоритм прекрасно работает с потоками, потому что данные обрабатываются индивидуально и по порядку, и граф задачи является динамическим конвейером, который представляет только параллелизм “поставщик-потребитель”.

## 4.3. Записи

Записи являются, возможно, разнородными коллекциями пар *поле-значение*. Так как Sisal 90 есть язык однократного присваивания, то записи конструируются за одну операцию, а не последовательностью присваиваний для каждого поля. Единственными операциями, определенными над записями, являются построение и доступ, однако можно расширить определения операторов, заданных в файле определений языка Sisal 90.

### *4.3.1. Построение записи*

Запись можно построить тремя способами: с помощью конструктора записи, типового конструктора и замещения. Пусть polar будет типом запись

```
type polar = record[r, theta: real];
```

Функция Polar_3

```
function Polar_3(x,y:real returns polar,polar,polar)
    record polar[r!x; theta!y],
    record[r,theta!x,y],
    polar{x,y}
end function  %Polar_3
```

возвращает три идентичные записи типа polar. Первые две конструкции являются примерами построения записи. Имена полей обязательны и должны стоять в том же порядке, в котором они стоят в определении типа. Имя поля можно ставить индивидуально, и за ним будет следовать его значение, либо имена могут представлять собой список, за которым следует выражение корректной арности и типа.

Третья запись в функции строится с помощью типовой конструкции. Каждый тип запись автоматически определяет типовый конструктор с тем же именем, что и имя типа. Конструктор работает с единственным значением

для каждого имени поля и присваивает полям значения в том порядке, в котором они стоят в определении типа.

Операция замещения строит новую запись, замещая один или более элементов в первоначальной записи. Следующее *выражение let* иллюстрирует эту операцию:

```
let
    x := 2.0;
    y := $pi/4.0;
    A := record[r,theta ! x,y]
in
    % удвоение величины вектора
      replace A[r ! 2*x],
    % отражение вектора относительно оси x-y
      replace A[theta ! y+$pi],
    % удвоение величины вектора и отражение его относительно оси x-y
      replace (replace A[r ! 2*x]) [theta ! y+$pi]
end let
```

Последнее выражение можно переписать в виде

```
replace A[r ! 2*x; theta ! y+$pi]
```

или

```
replace A[r,theta ! 2*x, y+$pi]
```

### *4.3.2. Доступ к полям записи*

Значения полей доступны с помощью dot-нотации, как и в других языках программирования. Например, если A — значение типа polar, тогда A.r и A.theta возвращают соответственно величину и угол A.

Dot-нотацию можно применять в любых выражениях и вызовах функций, которые возвращают записи. Например, функция

```
function reflect(A : polar returns polar)
    replace A[theta ! 2*$pi — A.theta]
end function
```

отражает A относительно оси x. Выражение reflect(A).theta возвращает угол отраженного вектора. К тому же выражение

```
(if 0<=A.theta<=$pi then A else reflect(A) end if).theta
```

является правильным, хотя и трудным для понимания. Заметим, что скобки вокруг *выражения if* необязательными и расставлены только для ясности.

## 4.4. Союзы

Союзы в языке Sisal 90 являются вариантом записи. Тип союз есть множество из одного или более тэгов. Каждый тэг имеет уникальные значение и тип. Ниже приведен пример декларации типа союз:

```
type scalar_numeric = union[i:integre; r:real; d:double];
```

Экземпляр типа союз имеет тэг и значение. Значение имеет тот же тип, что и тэг. Если A имеет тип scalar_numeric и его тэг — i, тогда его значение имеет тип — integer; если его тэг — r, то его значение имеет тип — real; если его тэг — d, то его значение имеет тип — double. Тэг не является полем в смысле записи, это атрибут времени исполнения, который идентифицирует тип значения союза.

Важно понимать разницу между союзами и множествами типов. Множество типов есть множество типов, в то время как союз является инкапсуляцией различных типов. Нужно рассматривать союз как контейнер. Если A и B имеют тип scalar_numeric, то они оба являются контейнерами одинакового типа. Выражаясь фигурально, их можно представлять себе как картонную коробку или металлический барабан. Контейнер scalar_numeric может хранить три типа “материалов”: integer, real и double. Тэг контейнера специфицирует тип хранимого материала.

Для союзов определены только операции построения и доступа к тэгу и значению, однако можно расширить это множество, данное в файле определений языка Sisal 90, и ввести специфические операции на союзах.

### *4.4.1. Построение союзов*

В языке имеется два способа построения союзов: с помощью конструктора союза и типового конструктора.

```
U1 := union scalar_numeric[i ! 10];
U2 := scalar_numeric{i ! 10}
```

Первое выражение является примером конструктора союза, где имя типа и значение тэга обязательны. Как и в случае записи, каждый тип союза определяет новый типовой конструктор с тем же именем, что и тип союза. Этот конструктор работает с двумя параметрами: тэгом и значением. Обе операции построения требуют, чтобы значение имело тот же тип, что и тэг.

### *4.4.2. Доступ к тэгу и значению союза*

Если тэг известен, то значение можно получить с помощью dot-нотации.

```
let
    u1 := scalar_numeric{i ! 11};
    u2 := scalar_numeric{r ! 22.2};
    u3 := scalar_numeric{d ! 33.3d0}
in
    U1.I, U2.R, U3.D   % 11, 22.2, 33.3d0
end let
```

При попытке получить значение по неправильному тэгу будет выдано ошибочное значение того же типа, что и тип неправильного тэга.

```
let
    u4 := scalar_numeric{i ! 11}
in
    u4.r   % error[real]; u4 имеет тэг i; тэг r имеет тип real
end let
```

Значение тэга нельзя достичь прямо. Извлечь его можно только с помощью предопределенной функции tag или управляющего *выражения case*. Первая является булевской функцией от двух аргументов: выражение типа союз и идентификатор. Если тэг выражения совпадает с идентификатором, функция возвращает true, иначе она возвращает false. Далее следует пример использования функции tag.

```
if tag(u1, i) then
    “An integer value”
else
    “Must be something else”
end if
```

*Выражение case*, которое может выбирать из тэгов, начинается со слов case tag. Ниже приведен пример такого выражения:

```
case tag u1
    of i then u1.i
    of r then integer(u2.r)
    of d then integer(u3.d)
end case
```

Следует заметить, что ветви *выражения case* возвращают выражения одинакового типа вне зависимости от тэга, контролирующего ветвь. Ветви *выражения case*, которое работает с союзом, должны быть согласованы по арности и типу.

## 5. ЦИКЛИЧЕСКИЕ ВЫРАЖЕНИЯ

Циклические выражения являются основной частью большинства программ на языке Sisal 90. Они обеспечивают эксплуатацию большинства видов параллелизма на коммерческих мультипроцессорных системах. Sisal 90 содержит как параллельные, так и последовательные формы циклов, что позволяет закодировать итерационные алгоритмы для SIMD и MIMD архитектур. Синтаксис и семантика параллельной формы циклического выражения гарантируют надежность кода, которая заключается в отсутствие тупиков. Последовательная форма допускает циклические зависимости, сохраняя при этом семантику однократного присваивания.

Циклы кодируются с помощью *выражений for*. Общая форма этого выражения приведена ниже:

for циклическое предложение
returns предложение возврата end for

Циклическое предложение содержит управляющую конструкцию и тело цикла. Под первой подразумевается тест либо генератор диапазона, последнее является множеством связей имя-значение. Предложение возврата состоит из одного или более операторов редукции. Каждый оператор редукции работает над значениями, определенными экземплярами тел цикла, и преобразует их в скалярное значение либо строит из них агрегатную структуру данных. Sisal 90 предопределяет десять операторов редукции и позволяет программисту определять свои собственные редукции.

В языке имеется четыре формы *выражения for*, различающиеся первыми ключевыми словами, ниже приведены их различия:

1) for while — последовательное выражение. Тело цикла управляется пре-тестом и выполняется, пока тест истинен.
2) for repeat — последовательное выражение. Тело цикла управляется пост-тестом, таким образом, первый экземпляр тела цикла всегда выполняется. Тестом является *выражение while*.
3) for initial — последовательное выражение. Выражение разделяет первую инициализирующую итерацию и тело цикла. Эта первая итерация всегда вычисляется, за ней вычисляется ноль или более итераций тела цикла. Тестом является *выражение while*, которое может либо предшествовать, либо следовать за телом цикла.
4) for — параллельное или последовательное выражение. Если ключевое слово old присутствует либо в циклическом предложении, либо в предложении возврата, тогда выражение является

последовательным, иначе оно параллельное. Выражения тела цикла могут вообще не вычисляться.

Поскольку тело цикла и предложение возврата аналогичны для всех четырех форм цикла, они обсуждаются сначала, а затем будут описаны специфические особенности каждой формы в отдельных разделах.

### 5.1. Тело цикла

Тело цикла содержит множество связей имя-значение и выполняется аналогично первой части *выражения let*, обсуждавшегося в главе 2.1. Тело может быть пустым, как в следующем примере:

```
for i in 1,3
returns array of i*i
end for
```

Правила связывания имен, стоящих в теле цикла приведены ниже:

1. Имя в теле цикла может связываться только один раз.
2. Каждое имя, используемое в правой части оператора присваивания, должно быть определено прежде в циклическом предложении или во внешней области действия, видимой для *выражения for*.

Имена, задействованные в *выражении for*, делятся на два класса:

1. Константы цикла — это имена, определяемые во внешней области действия и импортируемые в выражение, они не переопределяются в циклическом предложении.
2. Циклические имена — это имена, которые связываются с значениями в циклическом предложении. Эти имена можно разделить на три подкласса:
   a) диапазонные — имя связывается в генераторе диапазона;
   b) циклически локальные — эти имена стоят в левых частях операторов присваивания и не могут находиться в циклическом предложении или предложении возврата после ключевого слова old. Все имена, связываемые в теле параллельного *выражения for*, являются циклически локальными;
   c) циклически зависимые — либо диапазонные, либо циклически локальные имена, которые находятся в циклическом предложении или в предложении возврата после ключевого слова old.

Ключевое слово old может предшествовать любому циклическому имени, что обеспечивает циклически зависимые зависимости. Значением имени old name является значение имени name на предыдущей итерации. Переопределение значения имени name на текущей итерации не разрушает старое значение, оба сосуществуют вместе, и на них ссылаются соответственно, как на name и old name. Рассмотрим *выражение for initial*, которое реализует итерацию Ньютона-Рапсона (Newton-Raphson) для значений x, initial_guess и epsilon, определенных во внешней области.

```
for initial
    guess := initial_guess
repeat
    guess := (old guess + x / old guess) * 0.5d0
while abs(guess —  old guess) >= guess * epsilon
returns value of guess
end for
```

Инициализирующая итерация (код, стоящий между ключевыми словами for initial и repeat) определяет первое значение циклического имени guess. Каждая последующая итерация заново связывает циклическое имя как функция значения этого имени на предыдущей итерации. Заметим, что оба значения, новое и старое, используются в *выражении while* в конце циклического предложения.

Для каждой формы *выражения for* правила появления слова old, определения значения имени old name на первой итерации и время нового связывания имени old name объясняются в деталях в последующих разделах.

### 5.2. Предложение возврата

Предложение возврата содержит одну или более операций редукции, возможно разделенных запятыми. Тип и арность операции редукции определяется типом и арностью *выражения for*. Определенные пользователем редукции могут возвращать кратные значения различных типов, например, максимальное значение массива и индекс, где оно впервые встретилось.

Повторяющееся выполнение тела цикла порождает последовательность связей имя-значение. Операция редукции преобразует эту последовательность в одно или более скалярных значений или собирают их вместе для построения одной или более агрегатной структуры. Например, редукция product of x возвращает произведение всех связанных с x значений. Порядок порождения упомянутой последовательности и, таким образом, порядок редуцирования определяются семантикой цикла и являются детерми-

нированными. Обычно i-ая итерация тела цикла определяет i-ое значение в последовательности. При реализации языка можно игнорировать эту последовательность только для редукций, которая является математически ассоциативной и коммутативной.

### *5.2.1. Предопределенные редукции*

Sisal 90 предопределяет десять операций редукции. Предположим, x является циклическим именем, тогда:

- value of x — возвращает последнее значение x;
- sum of x — возвращает сумму значений x;
- product of x — возвращает произведение значений x;
- least of x — возвращает минимальное значение x;
- greatest of x — возвращает максимальное значение x;
- array of x — возвращает массив, построенный из значений x;
- array_*n*d of x — возвращает n-мерный массив, построенный из значений x;
- stream of x — возвращает поток, сформированный из значений x;
- stream_*n*d of x — возвращает n-мерный поток, сформированный из значений x;
- catenate of x — возвращает массив или поток, построенный конкатенацией значений x (x должен быть либо массивом, либо потоком).

Редукции array_*n*d of x и stream_*n*d of x могут находиться только в форме for выражения for и обсуждаются в главе 5.6.2. Каждая редукция возвращает значение по умолчанию, если не один экземпляр тела цикла не выполняется.

Параллельная и итеративная двойственность *выражения for* и универсальность предложения дают языку Sisal 90 большую мощность и возможность кодирования общих параллельных конструкций программирования. Следующее выражение иллюстрирует синтаксис и семантику предопределенных операций редукции. В цикле выполняются три итерации: i = 1, 2 и 3.

```
for i in 1,3
returns value    of i          % last value = 3
        sum      of i          % 1+2+3=6
        product  of i          % 1*2*3=6
        least    of i          % min(1,2,3)=1
        greatest of i          % max(1,2,3)=3
```

```
array    of i              % [1!1,2,3]
array(0) of i              % [0!1,2,3]
stream   of i              % stream[1,2,3]
catenate of text[i]        % [S,I,S,A,L, ,I,S, ,N,I,C,E]
catenate(0) of text[i]     % [0!S,I,S,A,L, ,I,S, ,N,I,C,E]
end for
```

где text — есть строковый массив [“SISAL”, “IS”, “NICE”].

Семантика операций редукции, возвращающих массивы и потоки, более сложна, чем семантика остальных редукций. Операции всегда возвращают одномерные структуры с базовым типом type(i). Если i — целое, тогда редукция array of i вернет массив целых, если i — массив целых, тогда та же редукция вернет массив массивов целых. Число значений в последовательности определяет размер структуры. По умолчанию нижней границей массива всегда является 1, однако программист может использовать описание array(n) и catenate(n), где n — целое выражение, для установки значения нижней границы, отличной от 1.

В дополнение к имеющимся предопределенным операциям редукции Sisal 90 позволяет кодировать свои собственные редукции. Описание определенных пользователем редукций и их постановку в предложение возврата *выражения for* содержится в главе 6.2.

### *5.2.2. Фильтры*

В любой операции редукции можно использовать *фильтрующее выражение*. Ниже указана общая форма фильтра:

```
when булевское выражение
```

Фильтр действует как тест, включая циклические значения в редуцируемую последовательность, когда сам фильтр истинен. Таким образом, выражение

```
for i in 1,3
returns sum      of i when i mod 2 = 0
        product  of i when i mod 2 = 1
end for
```

возвращает значения 2 и 3. Первый фильтр включает в последовательность, редуцируемую операцией sum, только значение 2, в то время как второй фильтр удаляет из последовательности, редуцируемой операцией product, значение 2.

### *5.2.3. Циклы с нулевой прокруткой*

Возможно, что выражения for и for while (формы *выражения for)* ни разу не выполняют тело цикла. Такие выражения будем называть циклами с нулевой прокруткой (zero-trip loop) или пустыми циклами.

Когда тело цикла не вычисляется, операция редукции не может обработать последовательность циклических значений, однако каждый оператор должен вернуть какое-нибудь значение. В таких случаях с каждой операцией редукции ассоциировано возвращаемое по умолчанию значение. Ниже приведен список этих значений.

| Редукция | Значение по умолчанию |
|---|---|
| sum of x | 0 типа type(x) |
| product of x | 1 типа type(x) |
| least of x | максимальное число типа type(x) |
| greatest of x | минимальное число типа type(x) |
| array of x | пустой массив с компонентами типа type(x) |
| stream of x | пустой поток с компонентами типа type(x) |
| catenate of x | пустой массив или поток с компонентами типа type(x) |

Следующий код иллюстрирует значения, по умолчанию возвращаемые предопределенными редукциями в случае пустого цикла.

```
for i in 2,1
  x := array[1:i]
returns sum of i          % 0
        product of i      % 1
        least of i        % $maxint
        greatest of i     % $minint
        array of i        % array OneArr[1: ]
        array(5) of i     % array OneArr[5: ]
        stream of i       % stream OneStr[]
        catenate of x     % array OneArr[1: ]
    catenate(5) of x      % array OneArr[5: ]
end for
```

где OneArr и OneStr определены, соответственно, как array[integer] и stream[integer].

Значение, по умолчанию возвращаемое редукцией value of x, кажется, на первый взгляд, непонятным. Если x является константным выражением или константным циклическим именем, тогда редукция возвращает значение x. Если x — циклически локальное и не стоит в тесте, тогда возвращается ошибка типа type(x). Если x — циклически локальное и стоит в тесте или циклически зависимое, тогда возвращается значение, связанное с x во внешней области действия. Следующие две функции должны прояснить эту ситуацию.

```
function evolve(A:OneDim returns OneDim, integer)
  for while ~convergence(A) repeat
    A := advance(old A);
    B := change(A, old A)
  returns value of A
          value of B
  end for
end function   % evolve

function member(target: integer; A:OneDim returns boolean)
  let found := false
  in   for while size(A) > 0 & ~found repeat
       found := (target = old A[1]);
       A     := reml(old A)
    returns value of found
    end for
  end let
end function   % member
```

В первой функции A — циклически зависимое имя, B — циклически локальное. Более того B не стоит в тесте *выражения for while*. Если входное значение A удовлетворяет условию сходимости, выражение возвращает входное значение A и error[integer].

Во второй функции A — циклически зависимое имя, а found — циклически локальное. Более того, found стоит в тесте выражения. Таким образом, если вначале размер A есть 0, выражение вернет false.

Значения, возвращаемые по умолчанию редукциями array_*n*d и stream_*n*d, зависят от структуры генератора диапазона, который обсуждается в главе 5.6.2.

### 5.3. Выражение for while

*Выражение for while* является последовательной формой *выражения for*. Его общий вид приведен ниже:

for while булевское выражение repeat
  *тело цикла*
returns *предложение возврата*
end for

*Выражение while* управляет выполнением тела цикла и выполняется до вычисления тела. Тело цикла вычисляется до тех пор, пока выражение while истинно. Порядок выполнения теста, тела и предложения возврата следующий:

$$тест_0, тело_1, возврат_1, тест_1, тело_2, возврат_2, тест_2, \ldots$$

Если первое выполнение теста вернет ложь, то ни один экземпляр тела цикла не будет вычислен (цикл с нулевой прокруткой). В этом случае операции редукции выражения возвратят значения по умолчанию.

Перед любым циклическим именем может стоять ключевое слово old. Значением имени old name является значение, связанное с именем name на предыдущей итерации. Предположим, экземпляр тела цикла $body_{i-1}$, $i \geq 2$ связывает значение $x$ с именем *name,* тогда немедленно за выполнением теста $тест_i$ и как раз до выполнения экземпляра тела цикла $тело_i$ значение $x$ связывается с именем old name, а имя name становится неопределенным до следующего своего определения. Следом за вычислением теста $тест_0$ и перед выполнением экземпляра тела цикла $тело_1$ имя old name связывается с значением имени name из внешней области действия. Имена old name и name могут стоять в теле цикла и в предложении возврата, но в тесте может стоять только имя name. Это ограничение обязательно, поскольку значение имени old name не определено для теста $тест_0$.

Тела функций evolve и member, определенных в предыдущей главе, дают наглядный пример *выражения for while*.

### 5.4. Выражение for repeat

*Выражение for repeat* является последовательной формой *выражения for*. Общий вид этого выражения приведен ниже:

for repeat тело цикла
while булевское выражение

```
  returns предложение возврата
  end for
```

Порядок выполнения сегментов выражения следующий:

*тело*$_1$, *возврат*$_1$, *тест*$_1$, *тело*$_2$, *возврат*$_2$, *тест*$_2$, . . .

Так как тест вычисляется после выполнения тела цикла, один экземпляр тела всегда выполняется.

Перед любым циклическим именем можно поставить ключевое слово old. Значением имени old name является значение, связанное с именем name на предыдущей итерации. Предположим, экземпляр тела цикла $body_{i-1}$, $i \geq 2$ связывает значение $x$ с именем *name,* тогда немедленно за выполнением теста $тест_i$ и как раз до выполнения экземпляра тела цикла $тело_i$ значение $x$ связывается с именем old name, а имя name становится неопределенным до следующего своего определения. Перед выполнением первого экземпляра тела цикла имя old name связывается с значением имени name из внешней области действия. Как имя old name, так и имя name могут находиться в теле цикла, в предложении возврата и в тесте. В отличие от *выражения for while* тест впервые вычисляется после выполнения первого экземпляра тела цикла, поэтому можно позволить old-именам стоять в тесте.

В качестве примера *выражения for repeat* рассмотрим функцию evolve, закодированную с использованием этого выражения.

```
function evolve(A: OneDim returns OneDim, integer)
  for repeat
    A := advance(old A);
    B := change(A, old A)
  while ˜convergence(A)
  returns value of A
          value of B
  end for
end function   % evolve
```

Эта функция и функция, определенная в главе 5.1.3, возвратят одинаковые значения, если входное значение A не удовлетворяет критерию сходимости. Если это так, то в зависимости от того, влияет или нет advance на формулировку сходимости, совпадут или нет первые результаты, вторые результаты всегда будут различными. Предыдущая версия функции вернет error[integer], в то время как эта версия функции возвратит целое значение.

### 5.5. Выражение for initial

*Варажение for initial* является последовательной формой *выражения for*. Это выражение имеет две общие формы:

for initial
  *первая итерация*
while *булевское выражение* repeat
  *тело цикла*
returns *предложение возврата*
end for

и

for initial
  *первая итерация*
repeat
  *тело цикла*
while *булевское выражение*
returns *предложение возврата*
end for

Порядок выполнения различных частей первого варианта выражения следующий:

*первая итерация, тест*$_1$*, тело*$_1$*, возврат*$_1$*, тест*$_2$*, тело*$_2$*, возврат*$_2$*, . . .*

второго варианта выражения следующий:

*первая итерация, тело*$_1$*, возврат*$_1$*, тест*$_1$*, тело*$_2$*, возврат*$_2$*, тест*$_2$*, . . .*

Если тест располагается перед телом цикла, то может вычисляться нулевой экземпляр тела; но так как первая итерация вычисляется всегда, выражение никогда не вернет редукционное значение по умолчанию. Если тест расположен после тела цикла, то как минимум один экземпляр тела всегда выполняется.

Любому циклическому имени может предшествовать ключевое слово old. Значением имени old name является значение, связанное с именем name на предыдущей итерации. Обозначим первую итерацию через *тело*$_0$, и предположим, что *тело*$_{i-1}$ связало значение x с именем name, тогда непосредственно перед вычислением *тело*$_i$ значение x связывается с именем old name, и имя name становится неопределенным до момента своего следующего связывания. Перед выполнением первой итерации имя old name связывается с значением имени name из окружающей области действия. Оба

имени old name и name могут стоять в первой итерации, тесте, теле цикла и предложении возврата.

*Выражение for initial* разделяет первую итерацию и последующие итерации. Таким образом, оно естественно представляет такие итерационные вычисления, в которых инициирующие вычисления отличаются от последующих вычислений. Например, рассмотрим вычисление первых сумм, математически определенное следующим образом:

> Пусть X есть массив с нижней границей 1 и верхней границей n; вычислить массив Y по правилу $Y[i] = \sum_{j=1}^{i} X[j]$ для $1 \le i \le n$.

Заметим, что Y[1] = X[1] и Y[i] = Y[i-1] + X[i] для $2 \le i \le n$. Данные вычисления наилучшим образом представляются с помощью *выражения for initial*.

```
for initial
  i := 1;
  y := X[1]
while i < n repeat
  i := old i + 1;
  y := old y + X[i]
returns array of y
end for
```

В качестве второго примера рассмотрим функцию evolve, представленную ранее. Эту функцию можно закодировать следующим образом:

```
function evolve(A_in: OneDim returns OneDim, integer)
  for initial
      A := A_in;
      B := 0
  while ˜convergence(A) repeat
      A := advance(old A);
      B := change(A, old A)
  returns value of A
          value of B
  end for
end function  % evolve
```

Данная версия этой функции возвращает такие же результаты, что и предыдущие две версии, если A_in не удовлетворяет условию сходимости. Если это так, то первый результат A_in будет таким же, как и тот, что выдает версия с выражением for while, однако он может не совпадать с результа-

том версии с выражением for repeat. Второй результат 0 отличается от результата версии с выражением for while и может совпадать с результатом версии с выражением for repeat. Функция evolve иллюстрирует тонкие различия между этими тремя последовательными формами цикла. Необходимо использовать правильную форму циклического выражения и аккуратно кодировать тест и предложение возврата.

## 5.6. Выражение for

Форма *for выражения for* может быть как последовательным, так и параллельным выражением. На самом деле очень просто можно определить, является ли выражение последовательным или параллельным. Если где-нибудь в выражении присутствует ключевое слово old, то имеются циклические зависимости, и экземпляры тела цикла должны выполняться последовательно; иначе семантика выражения гарантирует независимость по данным экземпляров тела цикла, и потому они могут вычисляться параллельно и в любом порядке. Это означает, что каждое имя, используемое в правых частях равенств в теле цикла, обязательно определено ранее в цикле или во внешней области действия, видимой для *выражения for*. Таким образом, без ключевого слова old невозможно сослаться на значение, определенное в другом экземпляре тела цикла. Более того, имена, которым присваиваются результаты *выражения for*, считаются неопределенными, пока не завершится выполнение выражения, поэтому в теле цикла нельзя ссылаться на какой-либо результат или элемент результата *выражения for*.

### *5.6.1. Простые диапазоны*

Форма for *выражения for* управляется генератором диапазона. Генератор диапазона определяет множество целых с помощью нижней и верхней границ

```
for i in 1,5   % это множество {1,2,3,4,5}
```

или массив, или поток

```
for x in A     % пусть A массив с нижней границей 1 и размера n,
               % тогда диапазоном является множество
               % {A[1],A[2],. . .,A[n]}
```

Элементы диапазона рассматриваются по порядку, и для каждого из них вычисляется экземпляр тела цикла, т.е. экземпляр тела вычисляется для первого значения диапазона, затем для второго и т.д. Если экземпляры тела

цикла независимы по данным, то они могут выполняться параллельно и в любом порядке.

В общем, целый диапазон определяется следующим образом:

for *имя* in *первое значение, второе значение, шаг*

Шаг необязательно присутствует и может быть положительным и отрицательным. По умолчанию шаг равен 1. Ниже приведены примеры целых диапазонов:

```
for i in 1, 10, 2   % это множество {1,3,5,7,9}

for i in 10, 5, -1  % это множество {10,9,8,7,6,5}
```

Мощность диапазона, a, следовательно, и количество выполняемых экземпляров тела цикла, как для положительных, так и для отрицательных шагов, задается формулой:

*max(0, (второе значение — первое значение + шаг) / шаг)*

Если диапазон имеет нулевую мощность, то ни один экземпляр тела цикла не выполняется, и каждая операция редукции в предложении возврата возвращает значение по умолчанию.

Диапазон массива или потока выглядит следующим образом:

for *имя* in A at *список имен индексов*

где A — имя списка индексов. Часть at является необязательной; если она присутствует, то индексы элементов массива или позиции элементов потока отсылаются соответствующим им экземплярам тела цикла. Например, если A — массив с нижней границей 5 и верхней границей n, то диапазон

```
for x in A at i
```

генерирует n — 5 + 1 экземпляров тела цикла. Первый экземпляр имеет x=A[5] и i=5, второй, соответственно, x=A[6] и i=6 и т.д.

Предложение at обеспечивает простой механизм, который позволяет рассматривать не только самое внешнее измерение массива или потока. Если часть at пропущена или список имен индексов имеет мощность 1, тогда экземпляр тела цикла выполняется для каждого элемента массива или потока, т.е. рассматривается только структура самого внешнего измерения. Если мощность списка имен индексов равна n, тогда рассматриваются n внешних измерений массива или потока. Например, пусть A — это массив

```
array[ array[ array[111,112,112], array[121,122,123] ]
       array[ array[211,212,213], array[221,222,223] ]
```

```
    array[ array[311,312,313], array[321,322,323] ]
  ]
```

тогда диапазоны

```
for x in A at i
for x in A at i,j
for x in A at i,j,k
```

сгенерируют, соответственно, множества:

- из 3 элементов, рассматривается только внешнее измерение

```
{( array[ array[111,112,112], array[121,122,123] ], 1),
 ( array[ array[211,212,213], array[221,222,223] ], 2),
 ( array[ array[311,312,313], array[321,322,323] ], 3)
}
```

- из 6 элементов, рассматриваются первое и второе измерение

```
{( array[111,112,112], 1, 1), ( array[121,122,123] ], 1, 2),
 ( array[211,212,213], 2, 1), ( array[221,222,223] ], 2, 2),
 ( array[311,312,313], 3, 1), ( array[321,322,323] ], 3, 2)
}
```

- из 18 элементов, рассматриваются первое, второе и третье измерение

```
{(111,1,1,1},(112,1,1,2),(113,1,1,3),
 (121,1,2,1),(122,1,2,2),(123,1,2,3),
 (211,2,1,1),(212,2,1,2),(213,2,1,3),
 (221,2,2,1),(222,2,2,2),(223,2,2,3),
 (311,3,1,1),(312,3,1,2),(313,3,1,3),
 (321,3,2,1),(322,3,2,2),(323,3,2,3)
}
```

Если число измерений структуры меньше мощности списка имен индексов, то компилятор сгенерирует ошибку. Диапазон, мощность списка имен индексов которого равна n, является n-мерным диапазоном. Многомерные диапазоны обсуждаются в главе 5.5.2.

Два и более диапазонов могут быть скомбинированы с помощью dot-произведения. Dot-произведение соединяет в пары каждый элемент своего левого операнда с каждым элементом своего правого операнда, генерируя при этом простой диапазон картежей. Например, dot-произведение

```
for i in 1,5 dot j in 1,10,2
```

генерирует диапазон {(1,1), (2,3), (3,5), (4,7), (5,9)}. Если размеры диапазонов не совпадают, тогда меньший дополняется ошибочными значениями.

### *5.6.2. Многомерные диапазоны*

В языке Sisal 90 существует два вида многомерных диапазонов:

1. Диапазон массива или потока с предложением at с более чем одним измерением, или
2. Цепочка cross-произведений диапазонов целых, массивов или потоков.

Если генератор многомерного диапазона включает dot-произведения, то сначала вычисляются они, оставляя для вычисления диапазона только цепочку cross-произведений. Многомерный диапазон подобен многомерному массиву с доступом по целочисленному диапазону вместо дуплетов и триплетов, а в диапазонах массива и потока такой доступ применяется вместо индексных массивов.

Различные операции редукции по разному воспринимают многомерные диапазоны. Арифметические редукции (сумма, произведение, последний и наибольший элемент), конкатенация и определенные пользователем редукции рассматривают многомерный цикл как плоский диапазон, чье множество значений является композицией более простых диапазонов. Редукции массива и потока рассматривают многомерный цикл как множество гнезд циклов, один цикл на один терм в цепочке cross-произведений. Поскольку в *выражении for* предложение возврата может содержать любую комбинацию операций редукции, возможно применять оба варианта рассмотрения к одному циклу.

Сначала обсудим семантику многомерных диапазонов в терминах арифметических редукций, конкатенации и определенных пользователем редукций. Cross-произведение собирает в пары каждый элемент диапазона, представляющего левый аргумент, с каждым элементом диапазона, представляющим правый аргумент. Размер диапазона равен произведению размеров аргументов.

Рассмотрим примеры многомерных диапазонов. Предположим, что A есть массив [ [10, 20] [30, 40] ].
Диапазоном для выражения

```
for i in 1,3 cross j in 4,5 cross k in 7,8
```

является множество

{ (1,4,7), (1,4,8), (1,5,7), (1,5,8), (1,6,7), (1,6,8),
(2,4,7), (2,4,8), (2,5,7), (2,5,8), (2,6,7), (2,6,8),
(3,4,7), (3,4,8), (3,5,7), (3,5,8), (3,6,7), (3,6,8) }

Диапазоном для выражения

for i in 1,3 dot j in 4,5 cross k in 4,8

является множество

{ (1,4,4), (1,4,5), (1,4,6), (1,4,7), (1,4,8),
(2,5,4), (2,5,5), (2,5,6), (2,5,7), (2,5,8),
(3,6,4), (3,6,5), (3,6,6), (3,6,7), (3,6,8) }

Диапазоном для выражения

for i in 1,3 cross x in A

является множество

{ (1,array[10,20]), (1,array[30,40]),
(2,array[10,20]), (2,array[30,40]),
(3,array[10,20]), (3,array[30,40]) }

Диапазоном для выражения

for i in 1,3 cross x in A at k, l

является множество

{ (1,10,1,1), (1,20,1,2), (1,30,2,1), (1,40,2,2),
(2,10,1,1), (2,20,1,2), (2,30,2,1), (2,40,2,2),
(3,10,1,1), (3,20,1,2), (3,30,2,1), (3,40,2,2) }

Элементы последнего диапазона представляют собой картежи из четырех составляющих (i, x, k, l). Экземпляр тела цикла вычисляется для каждого элемента диапазона. В параллельном цикле экземпляры тела могут выполняться в любом, однако, чтобы получать детерминированный результат, предложение возврата должно уметь восстанавливать порядок в последовательности сгенерированных циклических значений. Следует отметить, что конкатенация всегда собирает циклические значения по порядку.

В некоторой степени, аналогично доступу к массиву можно определить более сложные многомерные диапазоны, используя имена, введенные в диапазоне, в другом диапазоне, стоящем позже по цепочке. Например, можно добавлять элементы из нижнего треугольного подмассива массива n*n, как в приведенном ниже коде:

```
for i in 1,n cross j in 1,i
returns sum of A[i,j]
end for
```

Редукции array и stream рассматривают многомерные *выражения for* как гнездо циклов. Например, выражение

```
for i in 1,3 cross j in 10,50,10
returns array of i+j
end for
```

рассматривается как выражение, приведенное ниже:

```
for i in 1,3 returns array of
  for j in 10,50,10
  returns array of i+j
  end for
end for
```

Внутренний цикл возвращает массив целых, а внешний цикл возвращает массив массивов целых. Данное выражение вернет следующий результат:

```
array[ array[11,21,31,41,51],
       array[12,22,32,42,52],
       array[13,23,33,43,53]
     ]
```

Аналогично, выражение

```
for x in A at i,j
returns array of f(i,j,x)
end for
```

эквивалентно выражению

```
for row in A at i returns array of
  for x in row at j
  returns array of foo(i,j,x)
  end for
end for
```

Для редукций array и stream размерность генератора диапазона определяет размерность результата. Например, если генератор диапазона представляет собой цепочку cross-произведений из трех термов, то результатом будет трехмерная структура. Размер i-го диапазона определяет размер i-го измерения структуры. В случае массива по умолчанию нижняя граница

массива и всех массивов-компонент равна 1. Нижние границы можно установить целым значением, отличным от 1, если после ключевого слова следует заключенный в скобки список целых выражений. Мощность списка может быть меньше или равной размерности диапазона, и границы присваиваются от внешнего цикла к внутреннему. Например, выражение

```
for i in 0,2 cross j in 0,2 cross k in 0,2
returns array(0,0) of i+j+k
end for
```

возвращает массив

```
array[0! array[0!  array[1! 0,1,2],  array[1! 1,2,3],  array[1! 2,3,4],
          array[0!  array[1! 1,2,3],  array[1! 2,3,4],  array[1! 3,4,5],
          array[0!  array[1! 2,3,4],  array[1! 3,4,5],  array[1! 4,5,6]
      ]
```

Другие примеры многомерных массива и потока в *выражении for* приведены ниже.

```
for x in A at i,j
returns array of x*x
end for
```

возвращает массив массивов с базовым типом type(x). Массив имеет size(A) элементов, и каждый элемент *i, 1≤i≤size(A)*, является массивом, размер которого эквивалентен size(A[i]).

```
for i in 1,3 dot j in 4,6 cross k in 1,10
returns array of i+j+k
end for
```

возвращает массив массивов целых. Массив имеет три элемента, и каждый элемент является массивом из десяти элементов.

```
for i in 1,3 cross j in 4,6 cross k in 1,10
returns array(2,5) of i+j+k
end for
```

возвращает массив массивов массивов целых. Размерность этого массива есть 3›‹3›‹10. Нижними границами по каждому измерению являются целые 2, 5, 1.

Редукции array_*n*d и stream_*n*d возвращают массив или поток с n измерениями в отличие от размерности генератора диапазона. Параметр n должен быть меньшим или равным размерности диапазона, иначе компилятор

сгенерирует ошибку типа. Эти редукции рассматривают многомерные выражения как гнездо циклов. Например, выражение

```
for i in 1,3 cross j in 1,3 cross
    k in 1,3 cross l in 1,3
returns array_2d of i*1000+j*100+k*10+l
end for
```

эквивалентно выражению

```
for i in 1,3 returns array of
    for j in 1,3 returns catenate of
        for k in 1,3 returns catenate of
            for l in 1,3 returns catenate of
                array[i*1000+j*100+k*10+l]
            end for
        end for
    end for
end for
```

Построенный массив является двумерным массивом с 3 строками и 27 столбцами.

```
array[ array[1111,1112,1113,1121,1122,1123,1131,1132,1133,
             1211,1212,1213,1221,1222,1223,1231,1232,1233,
             1311,1312,1313,1321,1322,1323,1331,1332,1333],
       array[2111,2112,2113,2121,2122,2123,2131,2132,2133,
             2211,2212,2213,2221,2222,2223,2231,2232,2233,
             2311,2312,2313,2321,2322,2323,2331,2332,2333],
       array[3111,3112,3113,3121,3122,3123,3131,3132,3133,
             3211,3212,3213,3221,3222,3223,3231,3232,3233,
             3311,3312,3313,3321,3322,3323,3331,3332,3333]
       ]
```

В общем, если диапазон имеет размерность *k*, то массивы и потоки, построенные *k-n+1* внутренними циклами, конкатенируются в строковом порядке. Размер n-го измерения есть произведение размеров этих диапазонов. Размеры первых n-1 измерений массива или потока эквивалентны соответственно размерам первых n-1 диапазонов. По умолчанию нижней границей массива и всех массивов-компонент является 1. В примере, представленном выше, k=4, n=2. Массивы, построенные 3 внутренними циклами, конкатенируются в строковом порядке для построения одномерного массива раз-

мера 27. Размер первого измерения эквивалентен размеру первого диапазона, а именно — 3.

Рассмотрение многомерных *выражений for* как гнезда циклов для операций редукции до массива или потока проясняет структуру массива или потока, если один или более циклов являются нулевыми. Пусть

```
type OneDim = array[integer];
type TwoDim = array[OneDim];
type ThrDim = array[TwoDim];
type ForDim = array[ThrDim];
```

Рассмотрим выражение

```
for i in a,b cross j in c,d cross
  k in e,f cross l in q,h
returns array of i+j+k+l
ens for
```

которое эквивалентно следующему

```
for i in a,b returns array of
    for j in c,d returns array of
        for k in e,f returns array of
            for l in g,h returns array of i+j+k+l
            end for
        end for
    end for
end for
```

Во всех случаях выражение возвращает массив типа ForDim. Если циклом с нулевой прокруткой является

- i, тогда результатом является array ForDim[];
- j, тогда результат есть одномерный массив размера b-a+1, и каждый элемент его есть пустой массив типа ThrDim;
- k, тогда результатом является двумерный массив размера (b-a+1)›‹(d-c+1), и каждый элемент есть пустой массив типа TwoDim;
- l, тогда результат есть двумерный массив размера (b-a+1)›‹(d-c+1)›‹(f-e+1), и каждый элемент есть пустой массив типа OneDim.

Теперь рассмотрим выражение

```
for i in a,b cross j in c,d cross
  k in e,f cross l in q,h
```

```
    returns array_2d of i+j+k+l
    end for
```

которое эквивалентно выражению

```
    for i in a,b returns array of
        for j in c,d returns catenate of
            for k in e,f returns catenate of
                for l in g,h returns catenate of
                    array[i+j+k+l]
                end for
            end for
        end for
    end for
```

Во всех случаях выражение возвращает массив типа TwoDim. Если первым пустым циклом является i, тогда результатом является array TwoDim[]; если пусты j, k, l, тогда результатом будет массив размера b-a+1, каждый элемент которого есть пустой массив типа OneDim.

### *5.6.3. Параллельные и последовательные for выражения*

Форма for *выражения for* может быть параллельной и последовательной. Если выражение содержит ключевое слово old, тогда оно является последовательным, иначе оно параллельное. Экземпляры тела цикла параллельного выражения могут вычисляться параллельно и в любом порядке. Поскольку в теле цикла невозможно сослаться на имена и результаты другого экземпляра тела цикла, тупиковые ситуации исключены.

Для умножение двух матриц A и B размера n›‹n удобно применить параллельное *выражение for*:

```
    for i in 1,n cross j in 1,n
      c := sum(A[i,1:n]*B(1:n,j])
    returns array of c
    end for
```

Экземпляры тела цикла двумерного диапазона независимы по данным и могут выполняться параллельно. Более того, векторное произведение само по себе является параллельным вычислением. Фактически, если n мало, то последовательные вычисления будут выполняться быстрее.

Последовательное *выражение for* содержит ключевое слово old. Оно не может стоять в генераторе диапазона, но может находиться в теле цикла или предложении возврата. Более того, оно может предшествовать только

циклическому имени. Значением имени old name на первой итерации является значение имени name из окружающей области действия.

Sisal 90 включает эту форму *выражения for* для упрощения последовательных циклических выражений, которые вычисляют фиксированное число идентичных итераций. Например, следующая функция является вариацией функции First_Sum для массивов с нижней границей 1 и размера n.

```
function First_Sum(X:array[real] returns array[real])
    for i in 2,n
        X := old X[i! old X[i-1] + old X[i] ]
    returns value of X
    end for
end function
```

На первой итерации имя old X связывается с значением X из окружающей области действия, т.е. с значением формального параметра.

В качестве второго примера рассмотрим функцию для вычисления полинома методом Горнера, закодированную на языке Sisal 90.

```
function Horner(coeff:array[double]; X:double returns double)
  let poly := 0.0d0
  in for C in coeff
       poly := X*old poly + C
    returns value of poly
    end for
  end let
end function % Horner
```

Каждый экземпляр тела цикла производит четыре связывания имени со значением:

1) имя old poly связывается со значением имени poly из внешней области действия на первой итерации и со значением имени poly на предыдущей итерации на второй и более поздних итерациях;
2) C связывается по одному разу с каждым значением из массива coeff;
3) X связывается с значением формального параметра;
4) имя poly связывается со значением выражения X*old poly + C.

Все перечисленные связи живы, пока выполняется тело цикла и предложение возврата. Это позволяет гарантировать свойство однократного присваивания языка Sisal 90. Если массив coeff пуст, то выражение вернет ошибку error[double].

## 6. ФУНКЦИИ

Функции являются составляющими блоками программ на языке Sisal 90. Функции языка математически чисты. Они преобразуют входные значения в выходные без видимых сторонних эффектов, при этом не сохраняя состояние между вызовами. Каждый вызов функции уникален и независим от какого-либо другого вызова. Отсутствие сторонних эффектов дает следующие преимущества: функции просты для понимания, т.к. данное множество входных значений всегда преобразуется в один и тот же результат, и компилятор может оптимизировать выражения, содержащие вызовы функций. Например, все заголовки функций могут быть заменены телами этих функций, что позволяет провести удаление общих подвыражений, удаление инвариантов цикла, слияние циклов и другие преобразования. Возможность сторонних эффектов, которые дают функции, в императивных языках программирования часто препятствует или ограничивает применимость этих важных, оптимизирующих код, преобразований.

Функция имеет ноль или более входных значений и возвращает одно или более значений возможно различных типов. Таким образом, функции языка Sisal 90 могут иметь кратную арность. Например, следующая функция

```
function smaller(i,j:integer returns integer,array[character])
  if      i<j  then   i, ”first”
  elseif  i>j  then   j, ”second”
  else                i, ”neither”
  end if
end function          % whict_is_smaller
```

имеет два целых значения в качестве входных и возвращает целое значение и массив литерных значений.

Функции языка Sisal 90 могут быть рекурсивными, а также могут являться функциями высшего порядка. Определение функции можно импортировать из других единиц компиляции или экспортировать в них. Более того, импортируемые функции могут быть написаны на других языках программирования, таких как Fortran, C и C++. Заметим, что Sisal 90 не гарантирует надежность функций, написанных на других языках программирования. Если такая функция вызывается из тела параллельного *выражения for* и работает с общими ресурсами, то не исключены тупики.

Все функции должны декларироваться до своего использования. Внутри компиляционной единицы функции располагаются по тексту выше своих

вызовов. Если пользователь предпочитает другой порядок расположения, или две функции взаимно рекурсивны, необходимо использовать декларацию вперед, т.е. предложение *декларации функции вперед*. Общая форма этого предложения такова:

forward function *имя*(*список аргументов* returns *список типов*)

Первый список дает имена и типы формальных параметров, второй — типы результатов функции. Пример этого предложения приведен ниже:

forward function foo(a,b:integer; c:real; d:OneDim returns OneDim)

Можно опустить имена формальных параметров и только перечислить их типы. Например, можно переписать предыдущее предложение в виде:

forward function foo(integer, integer, real, OneDim returns OneDim)

Такие допущения также возможны в предложении *глобальной декларации функции*, которое обсуждается ниже.

Чтобы импортировать имена функций из других единиц компиляции, необходимо использовать предложение *глобальной декларации функции*. Общая форма этого предложения для функций, написанных на языке Sisal 90, следующая:

global function *имя*(*список аргумент*в returns *список типов*),

а для функций, написанных на других языках программирования, приведена далее

global function *имя*(*список аргументов* returns *список типов*) in "*язык*" is "*имя*"

Предопределенные функции не декларируются. Каждая инсталляция компилятора включает файл определений, содержащий имена всех предопределенных функций. Эти функции могут быть закодированы на языке Sisal 90 или других языках программирования, или могут входить в системную библиотеку подпрограмм. Обычно такие библиотеки включают функции обращения типов, функции для манипуляции числами с плавающей точкой и комплексными значениями, функции для работы с битами, функции отображений и функции ввода/вывода. Sisal 90 обеспечивает гибкий и удобный способ декларации предопределенных функций и использования множеств функций для определенного класса задач. Имена этих функций предопределены, а не зарезервированы, т.е. пользователь может переопределить эти имена по своему желанию.

Для экспорта функций в другую единицу компиляции, необходимо перечислить их имена в предложении define в исходном файле. Эта декларa-

ция нужна компилятору для генерации информации для связывания (линковки). Например, для экспорта функций f, g и h необходимо в исходном файле поместить строку

```
define f, g, h
```

Функции, не включенные в декларацию define, являются локальными для данной единицы компиляции.

### 6.1. Простые функции

Простой называется функция, которая не является определенной пользователем редукцией (глава 6.2) или функцией высшего порядка (глава 6.3). Общий вид простой функции приведен ниже:

```
function имя(список аргументов returns список типов)
    выражение
end function
```

Первая строка является заголовком функции. Она декларирует имя функции, имена и типы формальных параметров и типы результатов данной функции. Число и типы формальных параметров должны быть согласованы с действительными параметрами каждого вызова функции. Кроме того, число и типы результатов должны быть согласованы с телом функции. Например, заголовок функции, которая перемножает комплексные матрицы, должен быть следующим:

```
function mmul(n,m:integer; A,B:array[array[complex]]
    returns array[array[complex]])
```

Функция имеет четыре параметра. Первый и второй параметр являются целыми, третий и четвертый параметры есть двумерные комплексные массивы. Функция возвращает один двумерный комплексный массив.

Телом функции всегда является только одно выражение. Арность выражения есть арность функции. Что вычисляет выражение, то возвращает функция. В качестве выражения выступает любое, поддерживаемое языком Sisal 90, выражение. Следует отметить, что можно усложнить код, используя let-выражение как тело функции. Например:

```
function mmul(n,m:integer; A,B:array[array[complex]]
    returns array[array[complex]])
  let
    C := for i in 1,n cross j in 1,m
         returns array of sum(A[i,1:m]*B[1:m,j])
```

```
        end for
    in C
    end let
end function   % mmul
```

В данном случае *выражение let* не является необходимым, эту функцию можно закодировать следующим образом:

```
function mmul(n,m:integer; A,B:array[array[complex]]
    returns array[array[complex]])
  for i in 1,n cross j in 1,m
  returns array of sum(A[i,1:m]*B[1:m,j])
  end for
end function   % mmul
```

Функция вызывается, как только ее имя встречается в исходном коде. Функции единичной арности могут быть непосредственно подставлены.

```
D := mmul(n,m,A,B) + C;
```

Когда функция встречается в правой части оператора присваивания, количество имен в левой части этого оператора должно совпадать с числом результатов функции. Можно отбросить результат, подставив вместо его имени подчерк в левой части оператора присваивания. Например, функция first_minimum пробегает по массиву скалярных значений и возвращает минимальное значение и индекс, где оно впервые встретилось. Предложение

```
_,i := first_minimum(X);
```

отбрасывает само минимальное значение и возвращает только его индекс в массиве.

### 6.2. Определенные пользователем редукции

Определенные пользователем редукции — это специальные функции, которые могут вызываться только из *предложения возврата выражения for*. Функции работают с последовательностью циклических значений, сгенерированных экземплярами тела *выражения for*, и возвращают один или более результатов возможно различных типов. Рассмотрим пример такой редукции и ее использование:

```
reduction forces(natoms:integer          % начальные значения
          repeat x,y,z:real; i,j:integer % потоковые значения
          returns array[real])           % результаты
  for initial
```

```
    force := array_fill(1,3*natoms,0.0)   % инициализация
  repeat
    i3 := 3*(i-1);
    j3 := 3*(j-1);
    force :=
    old force[i3  !old force[i3]  -x;        % force в x на i
              i3+1!old force[i3+1]-y;       % force в y на i
              i3+2!old force[i3+2]-z;       % force в z на i
              j3  !old force[j3]  +x;       % force в x на j
              j3+1!old force[j3+1]+y;       % force в y на j
              j3+2!old force[j3+2]+z;       % force в z на j
             ]
  returns value of force
  end for
end function                                % forces

function bonds(natoms:integer;pair_list:TwoDim returns OneDim)
  for i in pair_list[1] dot j in pair_list[2]
    x,y,z := bond_force(i,j);
  returns forces(natoms) of x,y,z,i,j
  end for
end function                                % bonds
```

Функция bonds накапливает атомные силы множества атомов по направлениям x, y, z. Тело *выражения for* вычисляет силы, генерируемые каждым сцеплением атома, и посылает ее значение в функцию-редукцию forces, которая накапливает силы каждого атома.

Отметим характерные для редукции черты:

1. Заголовок редукции начинается ключевым словом reduction, а не function.
2. Заголовок специфицирует имя редукции, ноль или более инициирующих значений и их типы, одно или более потоковых значений и их типы и один или более тип результата.
3. Телом редукции является либо *выражение for initial*, либо *выражение for repeat*.
4. Если это *выражение for initial*, то первая итерация является функцией только инициирующих значений, и только ключевое слово repeat разделяет первую итерацию от тела цикла.

5. Тело цикла вычисляется в точности один раз для каждого множества потоковых значений, сгенерированных телом *выражения for* при вызове функции, таким образом, управления редукцией не требуется.
6. Редукция value of — единственная допустимая в *предложении возврата*.
7. Если редукция есть *выражение for repeat*, то может вернуться только последнее значение имени “инициирующего значения”.

Отметим следующие наблюдения относительно вызовов функций-редукций:

1. Начальные значения следуют за именем редукции слева от ключевого слова of.
2. Начальные значения берутся из окружающей области действия.
3. Список потоковых значений есть перечисленный через запятую список циклических имен. Список завершается следующим за ним ключевым словом.

Далее приведен пример редукции, использующей *выражение for repeat*.

```
reduction my_sum(x:integer repeat z:integer returns integer)
  for repeat
    x := old x + z
  returns value of x
  end for
end reduction   % my_sum

function Array_plus_x(x:integer; A:OneDim returns integer)
  for z in A returns my_sum(x) of z end for
end function   % Array_plus_x
```

При первом выполнении тела цикла имя old x связывается со значением начального параметра.
Редукция может не иметь имен начальных значений и может возвращать кратные результаты, что демонстрирует следующий пример:

```
reduction first_minimum(repeat x:real; index:integer
                  returns real, integer)
  for initial
    min_index := $maxint;
    min_value := $max_real
  repeat
    min_value, min_index :=
      if x < old min_value then
```

```
        x, index
      elseif x = old min_value & index < old min_index then
        x, index
      else
        old min_value, old min_index
      end if
    returns value of min_value
          value of min_index
    end for
  end reduction   % first_minimum
```

Sisal 90 включает *предложения определения редукции вперед и глобальной редукции*. Их общая форма приведена ниже:

forward reduction *имя*(*список аргументов* repeat *список аргументов* returns *список типов*)
global reduction *имя*(*список аргументов* repeat *список аргументов* returns *список типов*)

Редукции наиболее часто стоят в *предложении возврата* параллельного *выражения for*. Однако, в отличие от предопределенных редукций пользовательские могут быть неассоциативными и некоммутативными. Ненадежные редукции принуждают либо к последовательному вычислению *выражения for*, либо собиранию последовательности значений, генерируемых выражением, в массивы и обработке массивов по порядку.

## 6.3. Функции как первостепенные объекты

Sisal 90 разрешает пересылку имен функций в качестве параметров, частичное вычисление функций и динамическое создание функций. Последнее обычно имеет место для функций “высшего порядка”, которые возвращают новую функцию как результат.

### *6.3.1. Функции как параметры*

Пользователь может пересылать имена функций в качестве параметров. Как пример рассмотрим общий метод обработки деревьев. Было бы неудобно кодировать отдельную функцию обработки дерева для каждого бинарного оператора. Вместо этого можно написать одну функцию, которая имеет два параметра: рабочий массив значений и имя применяемой бинарной функции.

```
type numeric = [integer, real, double]
type OneDim  = array[numeric]
type BinFunc = function[numeric, numeric returns numeric]
function Tree(X:OneDim; F:BinFunc returns numeric)
  for while size(X) > 1 repeat
    m := size(old X);
    X  := for i in 1, m-2, 2
          returns array of f(old X[i], old X[i+1])
          end for
          ||
          if m mod 2 = 0 then f(old X[m-1], old X[m])
          else                old X[m]
          end if
  returns value of X[1]
  end for
end function   % Tree
```

Можно использовать функцию Tree следующим образом:

```
% Вернуть наименьший вход, больший 0. Если оба входа
% меньше или равны 0, то вернуть 0.
fucntion least_positive(a,b:numeric returns numeric)
  if a<=0 then max(0,b)
  elseif b<=0 then a
  else            min(a,b)
  end if
end function  % least_positive
function main(X:array[real] returns real)
  Tree(X, least_positive)
end function  % main
```

### *6.3.2. Частичное применение функций*

Частичное применение функции преобразует функцию от n аргументов в функцию от m аргументов, при m<n, связывая значения n-m ее формальных параметров. Эта операция обеспечивает применение общих функций. Например, пусть f — функция от трех целых аргументов, тогда предложения

```
g := f(x,_,_);
h := f(_,y,z);
```

определяют две новые функции. Первая является функцией двух аргументов, она идентична функции f, только с первым параметром связано значение x. Вторая функция является функцией одного параметра, она также идентична f, но второй и третий параметры связаны со значениями y и z соответственно. Такие функции можно вызывать прямо в области действия ее имени.

Подчерк в списке параметров сигнализирует о частичном вычислении функции. Компилятор генерирует новый граф задачи, идентичный графу функции, и соединяет нужные входные дуги со специфицированными значениями. Если эти значения известны во время компиляции, компилятор просто подставляет их в граф для возможно дальнейшей оптимизации.

Рассмотрим функции:

```
function multiply_add(a,b:integer; X:OneDim returns OneDim)
  for z in X returns array of a*z+b end for
end function   % multiply_add
function bit_reversal(n:integer returns OneDim)
 for initial
   double_add_0 := multiply_add(2,0,_);
   double_add_1 := multiply_add(2,1,_);
   index := array[0]
 while size(index) < 2 ** n repeat
   index := double_add_0(old index)|| double_add_1(old index)
 returns value of setl(index,0)
end fucntion   % bit_reversal
```

Первая итерация *выражения for initial* во второй функции определяет две новые функции double_add_0 и double_add_1. Затем эти функции используются в теле этого же выражения. Так как связываемые значения являются константами, компилятор имеет возможность оптимизировать новые функции. Фактически, можно ожидать, что компилятор удалит операцию прибавления в функции double_add_0 и заменит операцию умножения в обеих функциях на операцию сдвига.

Частично применяемые функции можно отправить в библиотеку подпрограмм. Например, собрать в библиотеку все функции типа

```
type RelF = function(real returns real)
```

использующих правило Симпсона. Описание этого метода на языке Sisal 90 приведено ниже

```
function simpson(a,b:real; m:integer; F:RealF
      returns real)
  let delta := (b-a)/m
  in delta/3*
    for i in 0,m,2      % предположим m — четное
      x0 := a+i*delta;
      x1 := x0 + delta;
      x2 := x1 + delta;
    returns sum of (F(x0)+ 4*F(x1) + F(x2))
    end for
  end let
end function
```

где a и b — пределы интегрирования, m — число разбиений интервала. Как указано, подпрограмма принимает только функции типа RealF, любые действительно значные функции с другим числом параметров не интегрируются с помощью функции simpson. Например,

```
function PolyN(Coeff:array[real]; x:real returns real)
  for i in 0, size(Coeff) – 1
  returns sum of Coeff[i]*x**i
  end for
end function   % PolyN
```

Было бы очень дорого кодировать различные версии функции simpson для каждого типа действительно значных функций. Частичное вычисление решает эту проблему. Рассмотрим предложение

```
let PolyC := PolyN(Coeff,_)
in simpson(a,b,m,PolyC)
end let
```

Оно определяет новую функцию PolyC типа RealF, связывая первый параметр функции PolyN с массивом Coeff, а затем пересылает новую функцию в подпрограмму simpson.

### *6.3.3. Динамическое конструирование функций*

Sisal 90 позволяет конструировать функции динамически. Динамическая функция определяется *выражением function*. Общая форма этого выражения подобна функциональному определению. Например, предложение

```
G := function(x:real returns real)
       sqrt(sqrt(x**n))
     end function
```

определяет новую функцию G. Выражение fucntion создает новую область действия для всех имен, стоящих в списке формальных параметров и в левых частях определений, стоящих в теле функции. Все остальные имена импортируются из окружающей области действия, и значения связываются с ними во время компиляции или во время исполнения. В приведенном примере выражение function создает новую область действия для x и импортирует значение n из окружающей области действия.

*Выражение function* может играть роль тела другой функции или *выражения for*. Последнее обеспечивает мощные динамические возможности. Например, необходимо вычислить значение выражения $x^{(1/2)^{**n}}$, где n — больше нуля и известно только во время компиляции. Можно закодировать эту функцию следующим образом:

```
for i in 1,n
  x := sqrt(old x)
returns value of x
end for
```

но если постоянно вызывать эту функцию, можно дойти до переполнения цикла. Более эффективная альтернатива — написать выражение, которое конструирует функцию, эквивалентную гнезду вызовов функции квадратного корня глубины n.

```
sqrt_N := for initial
          i := 1;
          f :=  function(x:real returns real)
                  sqrt(x)
                end function
        while i < n repeat
          i := old i + 1;
          f :=  function(x:real returns real)
                  sqrt(old f(x))
                end function
        returns value of f
        end for
```

Для n=3 предложение возвращает выражение function следующего вида:

```
function(x:real returns real)
  sqrt( sqrt( sqrt(x) ) )
end function
```

### 6.4. Функция main

Язык Sisal 90 определяет специальную функцию main для осуществления первоначального контроля и получения программных параметров. По умолчанию имя этой функции main.

Функция main является точкой входа в программу. Аналогично всем функциям она может иметь ноль или более входов и возвращает один или более результатов возможно различных типов. Тип type set не может быть типом формальных параметров этой функции. Это ограничение позволяет компилятору языка Sisal 90 разрешать типы большинства имен, избегая необходимости дорогостоящего разрешения типов во время исполнения. Для автономных программ значения формальных параметров считываются со стандартного входного устройства, и результаты записываются на стандартное выходное устройство. Входные и выходные значения разделяются пробелами или запятыми и имеют тот же порядок, в каком они стоят в заголовке функции.

Функция main уникальна тем, что она может определять глобальные имена. Строгая приверженность к функциональной семантике требует, чтобы функции зависели только от своих входных значений, однако это ограничение приводит к слишком длинным заголовкам функций и может потребовать пересылки имен вниз на много уровней по дереву вызовов, чтобы сделать их доступными отдаленному вызову. Sisal 90 ослабляет это ограничение. Он позволяет функции main определять и различать среди своих формальных параметров глобальные имена. Эти имена глобально доступны, т.е. на них можно сослаться в любой области действия. Так как эти имена являются специальными, им предшествует символ $.

Следующая программа

```
define  main
type OneDim = array[real];
global function rest_of_program(i:integer returns real)

function main(i:integer; $density:OneDim returns real)
  let n := i*i;
      $version := “3.1.4”;
      $squares := for j in 1,n returns array of j*j end for;
```

```
    in rest_of_program(n)
    end let
  end function   % main
```

определяет три глобальных имени $density, $version и $squares. Они могут стоять в любом месте текста программы и всегда имеют следующие значения: строка [“3.1.4”], массив [1, 4, 9, … ,n*n] и значение второго формального параметра. Элементы массива доступны обычным способом, т.е. $density[k] или $squares[k].

Глобальные имена можно определять только в функции main. Значение может быть результатом вызова функции с одним ограничением: функция, вызывающая определение глобального имени, не может ссылаться на какое-либо определенное пользователем глобальное имя и не может вызывать функции, которые ссылаются на глобальные имена. Это ограничение позволяет избежать ссылки на глобальное имя до его определения и предотвращает замкнутые определения.

Инсталляция языка Sisal 90 позволяет определить предопределенные имена, которые работают и располагаются так же, как глобальные имена. Например, $maxint, $minint, $pi и $e. Каждая инсталляция компилятора включает файл определений, в котором перечисляются эти имена. Этот файл можно редактировать для целевого применения.

## СПИСОК ЛИТЕРАТУРЫ


1. **Nikhil R.S.** Id Language Reference Manual. Version 90.1 // Computation Structures Group Memo 284-2, Laboratory for Computer Science. — M.I.T., 1991.
2. **Ranelletti J.E.** Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages // PhD thesis, University of California at Davis, Computer Science Department. — Devis, CA, 1987.
3. **Cann D.C.** Compilation Techniques for High Performance Applicative Computation // Technical Report CS-89-108. — Colorado State University, 1989.
4. **Skedzielewski S.K., Welcome M.L.** Data flow graph optimization in IF1 // Lect. Notes Comput. Sci. — 1985. — Vol. 201. — P.17-34.
5. **McGraw J.R. et.al.** Sisal: Striams and Iteration in a Singl Assigment Language. Language Reference Manual: Vertion 1.2. // Lawrence Livermore National Laboratory Manual M-146. — Livermore, CA, 1985.
6. **The Sisal 2.0 Reference Manual** // Bohm A.P.W., Oidenhoeft R.R., Cann D.C., Feo J.T. — Livermore, CA, 1991. — (Prepr. / Lawrence Livermore National Laboratory; UCRL-MA-109098).

7. **Евстигнеев В.А., Городняя Л.В., Густокашина Ю.В.** Язык функционального программирования Sisal // Интеллектуализация и качество программного обеспечения. — Новосибирск, 1994. — С. 21-42.
8. **Feo D.T., Miller P.J., Skedzielewski S., Denton S.M., Solomon C.J.** Sisal 90 // Proc. High Performance Functional Computing., April 9-11, 1995. — Denver, Colorado, 1995. — P. 35-47.
9. **Feo D.T., Miller P.J., Skedzielewski S., Miller P.J., Denton S.M.** Sisal 90 User's Guide. // LLNL, Livermore, CA, 1995. — Draft 0.96.

**Бирюкова Юлия Владиславовна**

# SISAL 90
# РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

**Препринт**
**72**

Рукопись поступила в редакцию 15.12.99
Рецензент Л. В. Городняя
Редактор З. В. Скок

---

Подписано в печать 10.04.2000
Формат бумаги 60 × 84 1/16 Объем 4.8 уч.-изд.л., 5.3 п.л.
Тираж 50 экз.

---

ЗАО РИЦ "Прайс-курьер" 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6