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

А. П. Стасенко, А. И. Синяков

БАЗОВЫЕ СРЕДСТВА ЯЗЫКА SISAL 3.1

Препринт
132

Новосибирск 2006

В работе описывается синтаксис и семантика новой версии потокового языка программирования Sisal 3.1, являющегося входным языком системы функционального программирования, разрабатываемой в рамках проекта ПРОГРЕСС. Язык Sisal 3.1 получен путем упрощения, улучшения, расширения и уточнения пользовательского описания языка Sisal 90, а также использования идей языка Sisal 3.0 и элементов объектно-ориентированного подхода. К элементам объектно-ориентированного подхода в языке Sisal 3.1 относятся пользовательские типы, переопределение операций и перегрузка функций.

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

Alexander Stasenko, Alexey Sinyakov

BASIC MEANS OF THE SISAL 3.1 LANGUAGE

Preprint
132

Novosibirsk 2006

This work describes syntax and semantics of the new version of dataflow programming language Sisal 3.1, being the source language of the system of functional programming (SFP) developed as a part of the project PROGRESS. Language Sisal 3.1 is obtained by simplification, improvement, expansion and specification of the Sisal 90 language user manual. It also uses some ideas of Sisal 3.0 language and elements of the object-oriented approach. New user types, operations redefinition and overloading of functions are elements of the object-oriented approach in Sisal 3.1 language.

ПРЕДИСЛОВИЕ

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

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

Препринт содержит описание базовых средств текущей версии входного языка SISAL 3.1 системы функционального программирования SFP, работа над которой ведется в лаборатории конструирования и оптимизации программ Института систем информатики СО РАН имени А.П. Ершова.

Цель проекта – предоставить прикладному программисту на его рабочем месте удобную среду для разработки функциональных программ, предназначенных для последующего исполнения на параллельных вычислительных системах, доступных через телекоммуникационные сети. В рамках этой среды программист получает возможность, с одной стороны, создавать и отлаживать программу без учета целевой параллельной архитекту-

ры, а с другой – производить настройку отлаженной программы на ту или другую целевую параллельную архитектуру для достижения высокой эффективности исполнения разработанной программы на супервычислителе.

Для более гибкого применения оптимизирующих преобразований на этапе трансляции и эффективного использования программ предполагается поддерживать возможность создания на языке SISAL 3.1 аннотированных программ, т.е. таких программных текстов, в которых содержатся как собственно программные части (базовые программы), так и спецификации контекстов их применений (аннотации или прагмы). Аннотации, специфицирующие контекст, позволяют пользователю управлять настройкой программы на супервычислитель и могут быть как утверждениями, так и директивами (следует отличать их от директив препроцессора). Аннотация-директива является средством неявного описания контекста, в то время как аннотация-утверждение позволяет специфицировать контекст явно. Утверждения могут использоваться для описания свойств не только объектов (например, число процессоров или диапазоны входных и выходных значений), но и конструкций аннотированной программы (например, отсутствие информационных зависимостей в цикле).

Синтаксически аннотация может быть помещена в любое место базовой программы, допустимое для комментария, и имеет вид формализованных комментариев. Семантика аннотаций в данном материале не рассматривается, но она такова, что игнорирование системой программирования аннотаций в процессе обработки SISAL 3.1 программы не должно влиять на результат получаемой программы, но может сказаться на ее эффективности.

Проф. В.Н. Касьянов

ВВЕДЕНИЕ

Потоковый язык программирования Sisal [1] является одним из самых известных потоковых языков промышленного уровня и позиционируется [2] как замена языка Fortran [3] для научных применений. Язык Sisal является результатом сотрудничества Ливерморской национальной лаборатории имени Лоренца, университета штата Колорадо, Манчестерского университета и корпорации DEC. Название языка Sisal – это аббревиатура выражения «Streams and Iterations in a Single Assignment Language» (потоки и итерации в языке однократного присваивания). Язык Sisal имеет следующие особенности, облегчающие переход с популярных императивных языков программирования:

- приближенный к языку Pascal [4] синтаксис;
- развитая система типов;
- явно выделенные циклические выражения.

Язык Sisal имеет следующие основополагающие свойства:

- математическая правильность функций (нет побочных эффектов);
- прозрачность ссылок имен, задающих значения, а не ячейки памяти;
- однократность присваивания.

Последняя спецификация языка Sisal версии 2.0 [5] датируется 1991 г. В 1995 г. также появилось пользовательское описание языка Sisal 90 [6, 7], не содержащее, однако, точных спецификаций языка.

В связи с этим актуальна задача разработки и спецификации новой версии языка Sisal, расширенной возможностями современных языков программирования. Новая версия языка Sisal 3.0 [8] была создана в 2001 г. в Институте систем информатики (ИСИ) имени А. П. Ершова СО РАН в качестве начальной версии входного языка системы функционального программирования SFP [9], разрабатываемой в рамках проекта ПРОГРЕСС. Язык Sisal 3.0 является развитием языка Sisal 90, ориентированного на серьезные научные применения. Нововведения языка Sisal 3.0 заключаются в возможности задавать отдельные части программы на императивном языке Си, расширенной поддержке модульности программ, возможности их предварительной обработки (preprocessing) и аннотирования для упрощения оптимизирующих преобразований.

Следующая версия языка Sisal 3.1, представленная в этой работе, является развитием языка Sisal 90 идеями улучшенной поддержки модульности и механизмами предварительной обработки, предложенными в языке Sisal 3.0. Вопросы, связанные с описанием мультиязыкового программирования

и аннотирования программ языка Sisal 3.0, в работе не рассматриваются и оставлены для внедрения в последующих версиях языка. Изменение языка Sisal 90 в языке Sisal 3.1 было обусловлено повышением удобства разбора и улучшением наглядности программ этого языка.

В работе приводится семантика языка Sisal 3.1 в нестрогом виде пользовательского описания, которое, по замыслу, претендует на полноту и непротиворечивость. Формальное описание синтаксиса языка Sisal 3.1, приведенное в приложении 1, является первым формальным описанием синтаксиса языка Sisal со времен языка Sisal 1.2. В приложении 2 демонстрируются элементы объектно-ориентированного подхода на примере модуля, задающего тип комплексного числа.

1. ОБЩИЙ ВИД SISAL 3.1 ПРОГРАММЫ

Программа на языке Sisal 3.1 (далее просто программа) является совокупностью модулей, содержащих определения функций с семантически выделенной функцией – точкой начала исполнения программы. Аргументы этой функции являются входными данными программы, а возвращаемые значения – ее выходными данными. Модулем программы (далее просто модулем) является один из перечисленных ниже объектов:

1) текст интерфейса и реализации модуля на языке Sisal 3.1;
2) внутреннее представление IR1;
3) специальным образом оформленный текст модуля языка Си++;
4) компонент (.DLL файл) единой объектной модели (COM), реализующий определенные интерфейсы объектов модуля.

Приведенную выше классификацию можно рассматривать и с точки зрения «жизненного цикла» типичного модуля (от первого пункта к четвертому). Далее рассматриваются синтаксис и семантика текста интерфейса и реализации модуля, понимаемого именно в этом смысле.

1.1. Элементы языка Sisal 3.1

Текст модуля задан символами уникода [10] в UTF-8 [11] кодировке.

1.1.1. Алфавит

Множество символов алфавита языка ограничено прописными и строчными буквами латинского алфавита, арабскими цифрами и специальными символами, приведенными в таблице 1 вместе с их десятичными ASCII

[12, 13] кодами. Следующие символы называются *пробельными* и разделяют другие конструкции языка: табуляция (десятичный ASCII код 9), перевод строки (код 10), вертикальная табуляция (код 11), новая страница (код 12), возврат каретки (код 13) и пробел (код 32). Остальные символы, не принадлежащие алфавиту языка, могут входить только в состав комментариев, символьных и строковых литералов.

Таблица 1

Специальные символы

Знак	!	"	#	%	&	'	(	)	*	+	,	-	.	/
Код	33	34	35	37	38	39	40	41	42	43	44	45	46	47
Знак	:	;	<	=	>	[	\	]	^	_	{	|	}	~
Код	58	59	60	61	62	91	92	93	94	95	123	124	125	126

1.1.2. Комментарии

Допустимы строковые комментарии, начинающиеся символами «//», и не вложенные друг в друга блочные (многострочные) комментарии, начинающиеся символами «/*» и оканчивающиеся символами «*/». Строчный комментарий эквивалентен символу перевода строки, а блочный комментарий рассматривается как пробельный символ при трансляции.

Примеры комментариев:

```
// Функция языка Sisal
function f(returns array[character])
  /* Тело функции */
  /* // Верно заданный комментарий */
  // // Верно заданный комментарий
  // /* /* Верно заданный комментарий */ */
  /* /* Неверно заданный комментарий */ */
  "Hello World"
end function
```

1.1.3. Идентификаторы

Идентификаторы задаются цепочкой букв верхнего регистра и букв нижнего регистра, отличных от букв верхнего регистра, десятичных цифр и знака подчеркивания. Идентификатор не может начинаться с десятичной цифры и состоять из единственного знака подчеркивания. Идентификаторы

используются для имен функций и значений, редукций, модулей и типов, образующих четыре синтаксически отдельных пространства имен.

Примеры идентификаторов:

```
A   a   Radian _   5degree
```

В приведенном тексте идентификаторами являются «A», «a», «Radian» и «f», а «_» и «5degree» – примеры неправильного задания идентификаторов. Причем идентификаторы «A» и «a» различны.

1.1.4. Ключевые слова

Существует (таблица 2) набор ключевых или зарезервированных слов, которые не могут быть идентификаторами. В тексте работы ключевые слова выделяются курсивом, а в текстах программ – жирным шрифтом.

Таблица 2

Ключевые слова

array	at	case	cross	declaration	definition	dot	else
elseif	end	error	false	for	forward	function	if
in	initial	is	let	nil	of	old	operation
record	reduction	repeat	replace	returns	set	stream	tag
then	true	type	union	uses	when	where	while

1.2. Препроцессор

Транслятор включает препроцессор, осуществляющий условную трансляцию, генерацию пользовательских предупреждений и ошибок, управление нумерацией строк и выделение именованных областей программы. В основе препроцессора лежит препроцессор языка C# [14]. Препроцессор языка реализован как часть лексического анализа и называется препроцессором для сохранения терминологии языков Си / Си++ [15, 16].

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

- директивы #define и #undef – определение и отмена определения символов условной трансляции (разд. 1.2.1);
- директивы #if, #elseif, #else и #endif – условная трансляция секций текста программы (разд. 1.2.2);

- директива #line – управление нумерацией строк, использующейся в сообщениях об ошибках и предупреждениях (разд. 1.2.3);
- директивы #error и #warning – генерация пользовательских ошибок и предупреждений (разд. 1.2.4);
- директивы #region и #endregion – дополнительная пометка секций текста программы (разд. 1.2.5).

Директивы препроцессора, лежащие в многострочных блочных комментариях, игнорируются. К одной директиве препроцессора, кроме директив #undef, #else и #endif, относятся также и строки, идущие следом и начинающиеся символами «‹возможные пробельные символы›# », где слова в угольных скобках – это метапонятия БНФ [17]. Содержимое последующих строк после символа «#» присоединяется к первой строке директивы.

1.2.1. Символы условной трансляции

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

1.2.2. Условная трансляция

Директива условной трансляции состоит из нескольких директив, описанных в нотации ISO Extended BNF [17], используемая часть которой кратко изложена в приложении 1:

```
"#if", булевское выражение,
  секция текста программы,
{ "#elseif", булевское выражение,
  секция текста программы
}, [ "#else",
  секция текста программы
], "#endif"
```

Булевским выражением является любое булевское выражение языка Sisal 3.1, содержащее имена символов условной трансляции и литералы *true* и *false*. Первое истинное булевское выражение директив #if и #elseif определяет транслируемую обычным образом секцию текста программы, возможно тоже содержащую вложенные директивы условной трансляции. Ес-

ли значения всех булевских выражений ложны, то транслируется секция текста программы, лежащая после директивы #else, если она присутствует. Транслируемая секция программы начинается со следующей после директивы строки или после окончания многострочного блочного комментария, начинающегося на одной строке с директивой препроцессора.

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

1.2.3. Нумерация строк

Директива #line имеет следующий синтаксис: «#line ‹номер строки› ‹имя файла›», где номер строки и имя файла заданы выражениями языка Sisal 3.1 целого и строкового типа. Директива меняет нумерацию и имя файла в возможных сообщениях об ошибках и предупреждениях текущего файла, начиная со следующей строки текста программы. Номер строки или имя файла можно опустить для сохранения его предыдущего значения. Если опущен номер строки и номер файла, то восстанавливаются значения по умолчанию, как если бы не было ни одной директивы #line до этого.

1.2.4. Диагностические директивы

Директива «#error ‹сообщение об ошибке›» генерирует ошибку трансляции с указанным сообщением, заданным выражением строкового типа языка Sisal 3.1. Сообщение об ошибке необязательно, и при его отсутствии будет сообщаться лишь о ее местоположении. Директива #warning аналогична директиве #error за исключением того, что вместо ошибки генерируется предупреждение.

1.2.5. Выделение секций текста программы

Тексту можно прикрепить пользовательскую пометку директивами:

```
"#region", [ пометка ],
  секция текста программы,
"#endregion", [ пометка ]
```

Необязательная пометка задается выражением строкового типа языка Sisal 3.1 и не обязана совпадать в директивах #region и #endregion. Секция текста программы транслируется обычным образом и также может содержать вложенные директивы #region … #endregion.

1.3. Модуль

1.3.1. Интерфейс модуля

Интерфейс модуля предназначен для определения типов, объявления операций, функций и редукций модуля, используемых другими модулями. Текст интерфейса модуля начинается с обязательного ключевого слова *declaration*, за которым должен следовать идентификатор имени модуля. Это имя используется другими модулями для получения доступа к объявлениям текущего интерфейса. Далее может находиться ключевое слово *uses*, за которым через запятую перечисляются идентификаторы имен модулей. В текущем интерфейсе модуля допускается использование типов, определенных в перечисленных модулях: «‹имя модуля› :: ‹имя типа›».

Далее в основной части текста интерфейса модуля находятся определения переименованных (разд. 2.4) и пользовательских (разд. 2.5) типов и объявления функций, операций и редукций, определяемых в реализации модуля. Точный синтаксис этих конструкций находится в приложении 1. Объявлением функции, редукции и операции является заголовок ее определения, где в квадратных скобках разрешается опускать имена формальных параметров, задавая только их типы. Функции без аргументов, ввиду отсутствия побочных эффектов, могут определять вычисляемые при компиляции константы программы.

Имена определяемых типов должны быть уникальны в пределах одного модуля. Имена функций и редукций одного модуля могут совпадать, если различны типы их формальных параметров. Знаки одинаковых операций должны иметь различные типы формальных параметров во всех используемых интерфейсах модулей.

Пример интерфейса модуля, в котором определяются тип и функция, возвращающая абсолютное значение:

```
declaration num
type num := real
function abs[num returns num]
```

1.3.2. Реализация модуля

Реализация модуля определяет операции, функции и редукции его интерфейса, а также локальные для нее определения типов, операций, функций и редукций. Текст реализации модуля всегда начинается ключевым словом *definition*, за которым идет идентификатор имени модуля. Далее

может находиться ключевое слово *uses*, за которым через запятую перечисляются идентификаторы имен модулей. В реализации модуля допускается использование типов, операций, функций и редукций, объявленных в интерфейсах указанных модулей: «‹имя модуля› :: ‹имя типа, функции или редукции›». Объявления и определения интерфейса модуля доступны в его реализации без указания его имени.

В основной части текста реализации модуля, расположенной далее, находятся определения функций, операций и редукций интерфейса модуля, описываемых в разд. 1.3.3, 3.4 и 5.3.2 соответственно. Основная часть модуля также содержит определения локальных типов, объявления (*forward*) и определения локальных функций, операций, и редукций. Точный синтаксис всех этих конструкций находится в приложении 1.

Далее приведен пример реализации модуля num:

```
definition num uses math
function abs(n: num returns num)
  math::abs(n:real)
end function
```

В этом модуле используется функция abs из модуля math, интерфейс которого имеет следующий вид:

```
declaration math
function abs[real returns real]
…
```

1.3.3. Определение функции

Определение функции имеет следующий вид:

```
"function", имя функции, "(",
  [ имена и типы аргументов ],
  "returns", типы результатов,
")", список выражений, "end", "function"
```

Часть определения функции до списка выражений называется заголовком функции. Размерность списка выражений должна совпадать с числом возвращаемых типов. Типы размерностей списка выражений должны также соответствовать возвращаемым типам, здесь и далее с точностью до их неявных преобразований, описанных в разд. 2.6. Список выражений здесь и далее состоит из разделенных запятыми выражений, не обязательно единичной размерности. Размерность списка определяется как сумма размерностей входящих в него выражений.

Пример функции, возвращающей знак числа:

```
function sgn(N: integer returns integer)
  if     N > 0 then  1
  elseif N < 0 then -1
  else               0
  end if
end function
```

В языках однократного присваивания отсутствует понятие переменной, заменяемое понятием обозначения идентификатором какого-либо программного фрагмента, уникального в пределах некоторой области видимости. Обозначение получает тип и величину сопоставленного элемента и может использоваться для дальнейшей передачи этих параметров в любую точку программы, обеспечивая копирование вычисленного объекта.

Нельзя ссылаться на имя до его объявления или определения. Определение имени может переопределяться для статически вложенной области. Не определенные в области имена импортируются из статически внешних областей. Определение функции вводит новую область имен значений с типами формальных аргументов функции.

2. ТИПЫ

Любой тип, кроме типа null, дополнительно содержит ошибочное значение, получаемое конструкцией «*error* [‹тип›]», что в дальнейшем при описании типов не указывается. Проверить, ошибочно ли значение типа, можно булевской операцией «*is error* (‹выражение›)», возвращающей значение *true*, если значение выражения ошибочно, и значение *false* иначе. Тип выражения можно получить конструкцией «*type* (‹выражение›)».

2.1. Простые (скалярные) типы

Простыми называются типы, не определяемые через другие типы.

2.1.1. Пустой тип

Тип null состоит из единственного значения *nil*. Операций, использующих или порождающих значение данного типа, не определено.

2.1.2. Булевский тип

Булевский тип boolean состоит из двух значений *true* и *false*.

2.1.3. Символьный тип

Тип character состоит из двухбайтовых символов уникода (Unicode-16). Литералы этого типа имеют вид знака символа в одинарных кавычках. Также допускаются литералы, приведенные в таблице 3, где ASCII-коды символов указаны как целые литералы языка Sisal 3.1.

В качестве примера приведем следующие символьные литералы:

```
'\n', 'S' = '\83' = '\h53' = '\o123'
```

Таблица 3

Коды обратной косой черты (escape-последовательности)

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

2.1.4. Целый тип

Тип integer является машинно-зависимым и обычно задает множество положительных и отрицательных целых чисел, допускаемых разрядностью машины. Обычно диапазон этого типа равен либо превосходит отрезок [- MaxInt() - 1, MaxInt()], где MaxInt() = 16#7FFFFFFF.

Значения типа задаются цепочкой десятичных чисел, для повышения читаемости которой везде, кроме ее начала, могут использоваться символы подчеркивания. Для вещественных чисел символы подчеркивания дополнительно не могут идти после десятичной точки. Знак числа, как и для вещественных чисел, считается унарной операцией, и поэтому между ним и числом допускается произвольное число пробельных символов. Целые литералы могут задаваться в произвольной системе счисления: «‹основание›#‹число›», где ее основание является числом от 2 до 36.

Например, целыми литералами являются:

```
2#1000000 = 8#100 = 16#40 = 36#1S = 64
```

2.1.5. Вещественные типы

Типы real и double задают машинно-зависимые множества действительных чисел соответственно одинарной и двойной точности, большей или

равной одинарной точности. Значения вещественных типов задаются десятичными литералами, отличающимися от целых литералов наличием точки и/или знаком экспоненты e (E) или d (D).

Вещественное число одинарной точности задается литералом простой или экспоненциальной формы. Простая форма является десятичным числом, разделенным знаком точки на две части, одна из которых может быть пуста. Экспоненциальная форма состоит из двух частей, разделенных буквой e или E. Левая часть – это десятичное число, возможно разделяемое на две части точкой, которая может стоять вначале. Правая часть – это необязательный знак минуса или плюса и десятичное число.

Вещественный литерал двойной точности отличается от вещественного литерала одинарной точности тем, что вместо буквы e (или E) используется буква d (или D). Для простой формы задания вещественного литерала двойной точности в конце числа должна стоять буква d (или D).

Далее рассмотрим примеры вещественных литералов:

```
5.0 = 5. = 0.5E1 = .5E1   // литералы типа real
5.0D = 5.D = 0.5D1 = .5D1 // литералы типа double
```

2.2. Массивы

Тип массива описывается как «*array* [‹тип элемента массива›]» и содержит конечные цепочки элементов одного типа с прямым доступом по их порядковому номеру, отсчитываемому от произвольного целого основания. Многомерность массивов эмулируется путем их вложенности.

Массив конструируется так: «*array* ‹тип массива› { ‹нижняя граница массива›; ‹элементы массива› }», где можно опустить ключевое слово *array*, тип, нижнюю границу (вместе со знаком точки с запятой) и элементы массива. Если тип массива опущен, но указан хотя бы один его элемент, то массив имеет тип «*array* [‹тип первого элемента массива›]». Если тип массива задан как «*array* [‹тип элемента массива›]», то нужно сократить одно ключевое слово *array*. Аналогичное правило сокращения двух повторяющихся ключевых слов действует и для конструкторов типов потока, записи и союза. Нижняя граница массива задается целым выражением и по умолчанию полагается равной единице. Если нижняя граница массива ошибочна, то и весь массив ошибочен. Если опущено ключевое слово *array*, то тип массива должен быть задан именем.

Строковые литералы рассматриваются как массивы символов «*array* [character]» с единичной нижней границей и задаются цепочкой символов, заключенной в двойные кавычки. Для задания символов строки допустимы

все обозначения таблицы 3. Единственная особенность связана с синтаксисом задания символа его кодом: если за ним не находится обратная косая черта, то необходим символ точки с запятой, обозначающий конец числа кода символа. Если непосредственно перед начальной кавычкой находится символ «@», то все специальные обозначения символов, кроме цепочки «\"», воспринимаются буквально. Последовательные строковые литералы, возможно расположенные на разных строках, склеиваются в один. Для массива типа «*array* [*T*]» определены встроенные операции таблицы 4.

Таблица 4

Встроенные операции над массивами

Имя	**Входы**	**Выходы**	**Описание**
size	*array* [T]	integer	Длина массива
liml			Нижняя граница массива
limh			Верхняя граница массива (liml + size)
reml		*array* [T]	Массив без первого элемента или «*error* [*array* [T]]», если массив пуст
remh			Массив без последнего элемента или «*error* [*array* [T]]», если массив пуст
addl	*array* [T], T		Массив с элементом, добавленным в начало
addh			Массив с элементом, добавленным в конец
setl	*array* [T], integer		Массив с новой нижней границей
fill	integer, integer, T		Массив с указанной нижней границей, размером и значением, его заполняющим

Над значением типа массив определена операция построения массива, полученного путем выбора или замены элементов массива: «‹массив› [‹операция выбора или замены›]». Последовательные операции выбора и замены можно помещать внутри одних квадратных скобок, разделяя их точкой с запятой.

2.2.1. Выражение выбора элементов массива

Выражение выбора элементов массива задается как «‹массив› [‹конструкция выбора›]». Конструкция выбора элементов массива состоит из цепочки операндов, разделенных ключевым словом *cross* (эквивалентного запятой) или ключевым словом *dot*. Каждый операнд задает индексы соот-

ветствующей размерности массива, причем левый операнд соответствует внешней размерности массива.

Операнд задается синглетом, являющимся унарным выражением целого типа, либо диапазоном: триплетом или массивом (потоком) целых чисел. Операндом также является «‹имя› *in* ‹диапазон›», где имя индекса можно использовать для задания зависимостей в операндах текущей конструкции выбора, стоящих правее следующего оператора *cross*.

Триплет – это операнд вида «‹начало› ! ‹конец› ! ‹шаг›», где начало, конец и шаг определяют числа некоторого диапазона и являются унарными выражениями целого типа. Все три выражения могут быть опущены и по умолчанию полагаются равными номеру первого, последнего элемента соответствующей размерности и sgn(‹конец› - ‹начало›) соответственно. Опускать ‹начало› и ‹конец› допускается только для первого или следующего после оператора *cross* диапазонов массива.

Если конструкция выбора задана списком синглетов, то выбираемый элемент массива очевиден, иначе, как показано далее, выражение выбора элементов массива заменяется циклическими выражениями. Представим конструкцию выбора выражения выбора *n*-мерного массива *A* как «D_1 … D_2 … D_n», где $D_{i=1…n}$ – это синглет или диапазон. Если f_1 … f_m являются порядковыми номерами диапазонов конструкции выбора, то выражение выбора представляется в виде цикла с многомерным диапазоном, где в соответствующих местах оператор *cross* заменен оператором *dot*:

for x_{f_1} **in** D_{f_1} **cross** x_{f_2} **in** D_{f_2} **cross** … x_{f_m} **in** D_{f_m}
returns array of `A [` d_1`, …,` d_n `]` **end for**

Имя x_{f_i} является любым новым именем, если D_{f_i} не имеет имени «‹имя *N*› *in* D_{f_i}», и равно имени *N* иначе. Выражение d_i равняется x_i, если *i* совпадает с каким-нибудь номером f_1 … f_m, иначе d_i равняется синглету D_i. Тем самым, произвольное выражение выбора элементов массива сведено к операции доступа к элементу массива по его индексу.

Для примеров, иллюстрирующих выбор и замену элементов массивов, определим одну матрицу размера 3 на 3 следующими способами:

```
type A2 = array [array [integer]]
a := {{1,2,3}, {4,5,6}, {7,8,9}}
a := array {{1,2,3}, {4,5,6}, {7,8,9}}
a := array [array[integer]]
  { {1,2,3}, {4,5,6}, {7,8,9} }
a := array [array[integer]]
```

```
  { 1; {1,2,3}, {4,5,6}, {7,8,9} }
a := A2 {{1,2,3}, {4,5,6}, {7,8,9}}
a := array A2 {{1,2,3}, {4,5,6}, {7,8,9}}
a := array A2 {1; {1,2,3}, {4,5,6}, {7,8,9}}
```

Примеры операций выбора элементов в массиве:

```
// элемент и строка матрицы
a[2,2] = 5; a[1,!] = {1,2,3}
a[1!3 dot 1!3] = {1,5,9} // диагональ матрицы
/* матрица с элементами, стоящими на пересечении 1-ой
и 2-ой строки и 1-го и 3-го столбца */
a[!2,!!2] = {{1,3}, {4,6}}
```

2.2.2. Выражение замены элементов массива

Выражение замены элементов массива задается как «‹массив› [‹конструкция выбора› := ‹конструкция замены›]». Конструкция выбора элементов массива определяет множество наборов индексов исходного массива. Набор индексов определяет заменяемый элемент массива. Результирующий массив получается из исходного массива путем замены указанных элементов.

Если конструкция выбора состоит из списка синглетов, то в качестве конструкции замены допускается указывать список выражений. Тип каждого выражения должен согласовываться с типом T размерности исходного массива, соответствующей последнему операнду конструкции выбора. Первое выражение заменяет элемент, определяемый конструкцией выбора, а последующие заменяют элементы, полученные путем увеличения на единицу значения последнего синглета в конструкции выбора. Если конструкция замены имеет тип T или тип массива с размерностью конструкции выбора и элементами типа T, то, как показано далее, выражение замены элементов массива выражается через циклические выражения.

Пусть конструкция замены не является списком синглетов, так как иначе выражение замены легко представимо в виде композиции операций поэлементной замены. Обозначим исходный массив именем A. Если конструкция замены задано значением типа элемента n-мерного массива A, то выражение замены – это цикл с многомерным диапазоном, где соответствующие операторы *cross* заменены операторами *dot*:

for x_{f_1} **in** D_{f_1} **cross** x_{f_2} **in** D_{f_2} **cross** … x_{f_m} **in** D_{f_m}
  A := **old** A [d_1, …, d_n := <конструкция замены>]

```
  returns value of A
end for
```

Если конструкция замены – это *k*-мерный массив элементов, имеющих тип элемента *n*-мерного массива *A*, то выражение замены выражается так:

```
let i1 := 1; in for x_f1 in D_f1 repeat
  i1 := old i1 + 1; i2 := 1;
  returns value of for x_f2 in D_f2 repeat
    i2 := old i2 + 1; i3 := 1;
    … returns value of for x_fm in D_fm repeat
      im := old im + 1;
      A := old A [
        d1, …, dn := <конструкция замены> [c1, …, ck]
      ]
      returns value of A
    end for
    …
  end for
end for end let
```

Для идущих последовательно *dot* операций «D_{f_i} *dot* D_{f_i+1} … *dot* D_{f_i+j}» в предыдущем выражении надо выделить следующие части:

```
for x_fi in D_fi repeat
  ii := old ii + 1; ii+1 := 1;
  returns value of for x_fi+1 in D_fi+1 repeat
    ii+1 := old ii+1 + 1; ii+2 := 1;
    … returns value of for x_fi+j in D_fi+j repeat
```

Заменяя эти выделенные части одним оператором *for*, получим:

```
for x_fi in D_fi dot
    x_fi+1 in D_fi+1 dot
    …
    x_fi+j in D_fi+j repeat
```

После такой замены останется ровно *k* значений $i_1 \ldots i_m$, равных индексам $c_1 \ldots c_k$. Тем самым, выражение замены элементов массива сведено к операции замены одного элемента массива по его индексу.

Здесь производится замена последнего столбца матрицы:

```
// замена на столбец из нулей
a[!,3 := 0] = {{1,2,0}, {4,5,0}, {7,8,0}}
// замена на столбец {9, 6, 3}
a[!,3 := {9,6,3}] = {{1,2,9}, {4,5,6}, {7,8,3}}
```

2.2.3. Векторные операции над массивами

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

Если типы элементов двух массивов допускают инфиксные операции, то они допустимы и для этих массивов, при этом порождается массив с нижней границей, равной единице, и элементами, полученными при поэлементном применении операции без учета нижних границ. Причем элементы меньшего по размеру массива дополняются ошибками.

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

Если векторная операция конфликтует с другой операцией над массивом, такой как выбор или замена элементов массива, то предпочтение отдается последней.

2.3. Другие составные (агрегированные) типы

Составными называются типы, определяемые через другие типы.

2.3.1. Поток

Тип потока описывается как «*stream* [‹тип элемента потока›]» и содержит цепочки последовательно доступных элементов одного типа. Поток конструируется так: «*stream* ‹тип потока› { ‹элементы потока› }». Если список элементов потока не пуст, то тип потока необязателен и по умолчанию определяется типом его первого элемента. Ключевое слово *stream* можно опустить, но тогда обязательно задание типа потока его именем. Поток типа «*stream* [*T*]» имеет встроенные операции таблицы 5.

Потоки поддерживают векторные операции, описание которых аналогично описанию векторных операций над массивами, приведенному в разд. 2.2.3, за исключением упоминаний о нижней границе. В случае инфиксной векторной операции над массивом и потоком получается поток.

Таблица 5

Встроенные операции над потоками

Имя	Входы	Выходы	Описание
size	*stream* [T]	integer	Длина потока
empty		boolean	Значение *true*, если поток пуст, и значение *false* иначе
first		T	Первый элемента потока или «*error* [T]», если поток пуст
rest		*stream* [T]	Поток без первого элемента или «*error* [*stream* [T]]», если поток пуст

Пример потока символов:

```
type SI = stream [character]
stream {'a', 'b', 'c'}
stream [character] {'a', 'b', 'c'}
SI {'a', 'b', 'c'}
stream SI {'a', 'b', 'c'}
```

2.3.2. Запись

Тип записи «*record* [‹поля записи›]» задает декартово произведение типов своих полей, к значениям которых имеется прямой доступ по их уникальному в пределах одного типа записи имени. Запись конструируется так: «*record* ‹тип записи› { ‹определения полей записи› }». Тип записи необязателен, и при его наличии список определений полей записи должен включать определения всех полей записи в порядке, определенном типом записи, что позволяет опускать имена полей. Если тип записи не указан, то он определяется именами и порядком заданных значений полей. Ключевое слово *record* можно опустить, но тогда обязательно задание типа записи его именем.

Есть операция получения записи из уже существующего значения типа запись: «‹запись› *replace* { ‹определения ее полей› }». Эта конструкция создает новую запись такого же типа, но с новыми значениями указанных

полей, перечисленных в их естественном порядке. Если запись является ошибочным значением, то порождается также ошибочное значение.

Значение поля записи можно получить так: «‹запись› . ‹имя ее поля›». Если запись представлена ошибочным значением, то результатом этой конструкции будет ошибочное значение типа требуемого поля записи.

Пример определения комплексного типа:

```
type complex = record [r, i: real]
record {r, i := 1.0, 2.0}
record {r := 1.0; i := 1.0}
record [r, i: real] {:= 1.0, 2.0}
record [r, i: real] {r, i := 1.0, 2.0}
record [r, i: real] {r := 1.0; i := 2.0}
complex {:= 1.0, 2.0}
complex {r, i := 1.0, 2.0}
complex {r := 1.0; i := 2.0}
record complex {:= 1.0, 2.0}
record complex {r, i := 1.0, 2.0}
record complex {r := 1.0; i := 2.0}
```

2.3.3. Союз

Тип союза «*union* [‹теги союза›]» аналогичен типу записи, исключая то, что не более чем одно значение его тега отлично от ошибочного значения. Можно опускать тип тега, если он равен null. Союз конструируется так: «*union* ‹тип союза› { ‹имя тега› := ‹его значение› }», где тип союза обязателен и должен содержать имя указанного тега. Ключевое слово *union* можно опустить, но тогда обязательно задание типа союза его именем.

Конструкция «‹союз› *tag* ‹имя тега›» истинна, если имя тега равно имени тега данного союза, ложна, если не равно, и ошибочна, если сам союз ошибочен. Конструкция «‹союз› . ‹имя тега›» равна значению, заданному именем тега союза, или ошибочному значению, если сам союз ошибочен.

Например, так может задаваться значение союза, задающее красный цвет светофора:

```
type rgy = union [red, green, yellow]
union [red, green, yellow] { red := nil }
rgy { red := nil }
union rgy { red := nil }
```

2.3.4. Функция

Тип функции задается как «*function* [‹типы аргументов› *returns* ‹типы результатов›]» и конструируется так:

- именем объявленной функции, если имя функции не перекрыто локальным именем и нет неоднозначности перегрузки;
- конструкциями «*function* ‹имя функции›», ««имя модуля› :: ‹имя функции›» или «*function* ‹имя модуля› :: ‹имя функции›» для устранения проблемы перекрытия локальным именем;
- конструкциями «*function* ‹имя функции› [‹типы аргументов›]» или «*function* ‹имя модуля› :: ‹имя функции› [‹типы аргументов›]» для полностью однозначного указания объявленной функции;
- конструкцией «*operation* ‹знак операции› [‹типы аргумента›]» для объявленной операции, кроме операции приведения типов;
- конструкцией «*operation* ‹знак операции› [‹тип аргумента› *returns* ‹тип результата›]» для объявленной операции приведения типов;
- конструкцией «*function* (‹имена и типы аргументов› *returns* ‹типы результатов›) ‹выражения› *end function*» для λ-функции.

Неоднозначность имени функции также разрешается оператором приведения типа «: ‹тип функции›». Тип функции имеет операцию ее вызова ««функция› (‹значения аргументов›)», дающую n-мерный результат значений с типами результатов функции. Если значение функции ошибочно, то все результаты операции будут ошибочны. Операция автоматически разрешает неоднозначность заданной функции на основании типов ее аргументов.

Если аргумент функции пропущен, то результатом операции вызова функции, называемой теперь «сужением» области определения функции, будет функция от пропущенных аргументов в порядке их следования и с результатами исходной функции. Остальные аргументы функции полагаются равными заданным значениям при вызове «суженной» функции. Операция также разрешает неоднозначность функции на основании типов ее аргументов, указываемых как «: ‹тип›» для пропускаемых аргументов.

Например, так может задаваться λ-функция, меняющая знак аргумента:

```
function fsgn(returns function[integer returns real])
  function(n: integer returns real) -n end function
end function
fsgn()(1) // вызов λ-функции, равный -1
```

2.4. Множество типов

Множество типов задается как «*set* [‹типы›]» или ключевыми словами *array*, *stream*, *record*, *union* и *function*, обозначающими произвольный тип массива, потока, записи, союза и функции соответственно. Вложенные множества типов раскрываются. Непосредственная зависимость множества типа от самого себя нивелируется. Множество типов, состоящее из одного типа, упрощается до этого типа, если он рекурсивно не зависит от данного множества типов. Пустое множество типов эквивалентно типу null. К *бесконечному* множеству типов, все типы которого рекурсивны относительно этого множества типов, добавляется тип null.

Значением множества типов является любое значение с типом, входящим в это множество. Ошибочных значений множества типов не существует. Множество типов не является обычным типом языка и служит только для поддержки полиморфизма, поэтому составной тип, зависящий от множества типов, тоже является множеством типов.

Для множества типов не определяется других операций, кроме операций его сужения и расширения и операций зависящего от множества типов составного типа. Операция сужения имеет вид: «‹значение множества типов *TS*› : ‹тип *T*›», где тип *T* принадлежит множеству типов *TS* и не является множеством типов. Если значение множества типов *TS* не принадлежит типу *T*, то значением этой операции будет ошибочное значение типа *T*. Операция расширения имеет вид: «‹значение типа *T*› : ‹множество типов *TS*›», где множество типов *TS* включает тип значения *T*. Значение типа *T* также может принадлежать множеству типов.

Определение переименованного типа выглядит как: «*type* ‹имя типа› = ‹тип›», где имя типа далее не переопределяемо. Использование имени переименованного типа семантически эквивалентно подстановке вместо него типа, стоящего в правой части определения. Языком определяется множество типов any, состоящее из всех *встроенных* типов, не являющихся пользовательскими: «*type* any = *set* [null, boolean, character, integer, real, double, *array*, *stream*, *record*, *union*, *function*]». Множества типов «*array* [any]» и «*stream* [any]» приводятся к эквивалентным типам *array* и *stream*.

В качестве примера определим множество типов, рекурсивно определяющее двоичное дерево:

```
type tree = set[null, integer, record[r, l: tree]]
```

2.5. Пользовательский тип

Определение пользовательского типа выглядит как: «*type* ‹имя типа› := ‹базовый тип›», где имя типа далее не переопределяемо. Множество значений пользовательского типа эквивалентно множеству значений базового типа, но для него изначально не определено никаких операций, кроме операции приведения типа. Операция приведения имеет вид «‹значение T_1› : ‹новый тип значения T_2›». Где пользовательский тип T_2 основывается на типе T_1 или пользовательский тип T_1 основывается на типе T_2. *Пользовательский тип CT основывается* на другом типе *T*, если тип *T* является базовым типом *BT* типа *CT* или пользовательский тип *BT* основывается на типе *T*. Для пользовательского типа разрешается определять новые операции, как описано в разд. 3.4.

В следующем примере определен пользовательский тип matrix:

```
type matrix := array [array [integer]]
```

2.6. Преобразования типов

Встроенные типы считаются эквивалентными при их *структурном совпадении*: если с точностью до разделителей совпадает их строковое представление после подстановки переименованных типов. Язык поддерживает неявные преобразования типов, образующие следующий порядок применения в случае неоднозначности:

1) отсутствие неявного преобразования;
2) преобразование integer в real (как операцией «: real»);
3) преобразование integer или real в double (как операцией «: double»);
4) пользовательское неявное преобразование (разд. 3.4);
5) преобразование значения типа *T* к пользовательскому типу, основывающемуся на типе *T*;
6) расширения множества типов, образующие порядок относительно включения формальных типов.

При возникновении неоднозначности выбора функции, операции или редукции, связанной с неявным преобразованием типов, среди множества доступных вариантов сигнатур выбирается та, порядок которой будет минимальным. Порядок множества сигнатур определяется как лексикографическое сравнение цепочек неявных преобразований из типов фактических параметров в типы формальных параметров.

Операции приведения типов перечислены в таблице 6. Для округления вещественного числа до ближайшего целого числа используется встроенная функция: «*function* round (set [real, double] *returns* integer)».

Таблица 6

Операции приведения типов

Приводимый тип	Операция	Примечания
double	: real	Потеря точности
real или double	: integer	Отбрасывание дробной части
character		Получение кода символа
boolean		Получение 0 для *false* и 1 для *true*
integer	: character	Символ по его коду или *error* [character]
	: boolean	Получение *false* для 0 и *true* иначе

Ниже приведены примеры явного и неявного преобразования типов:

```
function conversion(a, b, c: integer returns
                    integer, real, set[integer, real])
  3.1416 : integer // явное приобразование
  (a + b) / 2 // неявное преобразование к типу real
  // явное преобразование к множеству типов
  c : set[integer, real]
end function
```

3. АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ

3.1. Постфиксные операции

Постфиксные операции имеют вид «‹операнд› ‹операция›». Цепочка постфиксных операций вычисляется слева направо до начала вычисления префиксных операций. К постфиксным операциям относятся уже рассмотренные операции вызова или «сужения» функции, выбора или замены элементов массива, доступа к полю записи или союза, замены элементов записи и проверки тега союза.

Постфиксными операциями, например, являются:

```
f(1,2); f(1,) // вызов или сужение функции
2 : real      // преобразование типов
```

```
u tag a        // проверка тега союза
// создание новой записи с изменёнными элементами
r replace {a := 1; b,c := 2,3}
```

3.2. Префиксные операции

Префиксные операции имеют вид «‹знак операции› ‹операнд›». Цепочка префиксных операций вычисляется справа налево до начала вычисления инфиксных операций. Операции смены знака числа (знак минуса) и идентичности (знак плюса) определены для целых и вещественных значений. Операция логического отрицания (знак тильды) определена для булевского значения. При целочисленном переполнении возвращается значение *error*[integer]. Если операнд префиксной операции является ошибочным, то ее результат тоже будет ошибочным значением того же типа.

В следующем примере операции «~» и «-» являются префиксными:

```
function f(a: boolean returns integer)
  if ~a then 1 else -1 end if
end function
```

3.3. Инфиксные операции

Инфиксные операции имеют вид «‹операнд› ‹знак› ‹операнд›». Среди нескольких инфиксных операций раньше выполняются операции на более глубоком уровне вложенности арифметических скобок. Среди инфиксных операций одного уровня вложенности сначала выполняются операции с большим приоритетом (таблица 7). Лево-связываемые операции одного приоритета выполняются слева направо, а право-связываемые операции – справа налево. В таблице 7 также указана семантика инфиксных операций, сохраняемая и при их переопределении. Для операций сравнения сохраняется булевский результат, а для их цепочки – свойство склеивания операциями конкатенации, например: «A < B <= C» → «A < B & B <= C».

Если в результате выполнения операции / или % от целых операндов происходит деление на ноль, то возвращается *error*[integer], но для вещественного деления на ноль возвращается значение ±∞ стандарта IEEE-754 [18]. Если один из аргументов встроенной инфиксной операции ошибочен, то результатом операции тоже будет ошибочное значение.

В приведенном ниже примере демонстрируются некоторые инфиксные операции и их приоритеты:

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

Таблица 7

Неизменные свойства инфиксных операций

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

В таблице 8 приведены описания встроенных инфиксных операций.

Таблица 8

Встроенные инфиксные операции

Знак	**Описание операции ‹операнд A_1› ‹знак› ‹операнд A_2›**	
||	Конкатенация двух массивов или потоков. Для массивов порождается массив с нижней границей, равной единице.	
|	Дизъюнкция	Булевские операции оперируют с булевскими значениями, порождая булевскую величину.
^	Неэквивалентность	
&	Конъюнкция	
=	Равно	Операции определены для любых значений одинакового типа, кроме типа null, и для операции сравнения типов, получаемых как «*type* (‹выражение›)» и «*type* [‹тип›]».
~=	Не равно	
<	Меньше	Операции определены для символьных, целых и вещественных операндов.
>	Больше	
<=	Меньше либо равно	
>=	Больше либо равно	
-	Вычитание A_2 из A_1	Для символьных операндов возвращается целое значение разности кодов символов.
+	Сложение	• Целые операнды порождают целый результат. При делении и возведении в степень берется целая часть результата. • Операнды типа real порождают значение типа real. • Операнды типа double порождают значение типа double.
*	Умножение	
**	Возведение A_1 в степень A_2	
/	Деление A_1 на A_2	
%	Операция остатка от деления первого операнда на второй оперирует целыми числами, порождая целый результат.	

3.4. Пользовательские операции

Определение пользовательской операции имеет следующий вид:

```
"operation", знак операции, "(",
  [ имена и типы операндов ],
  "returns", типы результатов,
")", список выражений, "end", "operation"
```

В таблицах 9 и 10 указаны допустимые пользовательские операции.

Таблица 9

Пользовательские операции

Знак	**Примечания**	
+	$[A_1 \rightarrow R_1]$, где A_1 – это пользовательский тип. Префиксная операция	
-		
~		
=	$[A_1, A_2 \rightarrow$ boolean$]$, где A_1 или A_2 – это пользовательский тип. Если для типов A_1 и A_2 определены операции = и <, то для них неявно заданы операции <= и >=	Коммутативность. Неявно задает ~=
<		Неявно задает >
[]	$[A_1, A_2 \rightarrow R_1]$, где A_1 – это пользовательский тип. При применении операции, «E_0 [E_1, …, E_n]» эквивалентно «E_0 [E_1] … [E_n]»	
()	$[A_1, \ldots, A_n \rightarrow R_1, \ldots, R_m]$, где A_1 – это пользовательский тип	
. ‹id›	$[A_1 \rightarrow R_1]$, где A_1 – это пользовательский тип	
:	$[A_1 \rightarrow R_1]$, где A_1 или R_1 – пользовательский тип. Приведение типов	
ε	$[A_1 \rightarrow R_1]$, где R_1 – пользовательский тип. Неявное приведение	

Таблица 10

Инфиксные арифметические пользовательские операции

Знак	+	*	&	|	^	-	/	%	**	||
Замечания	$[A_1, A_2 \rightarrow R_1]$, где A_1 или A_2 – это пользовательский тип									
	Коммутативность по умолчанию									

Постфиксные операции замены элементов записи и проверки тега союза не переопределяются ввиду особенностей их синтаксиса. Если не определена пользовательская операция равенства двух одинаковых пользовательских типов, то автоматически используется операция равенства (и неравен-

ства) для типа, лежащего в основании пользовательского типа. *Тип GT лежит в основании пользовательского типа CT*, если тип *CT* основывается на типе *GT* и тип *GT* не является пользовательским типом.

Например, определим сложение для пользовательского типа num:

```
operation +(a, b: num returns num)
  a:real + b:real
end operation
```

4. ВЫРАЖЕНИЯ

4.1. Выражение *let*

Выражение *let* определяет новую область и множество ее имен, используя их для вычисления списка выражений своих результатов:

```
"let", определения имен,
"in", результаты, "end", "let"
```

Определения имен содержат разделенные точкой с запятой определения, содержащие левую и правую части, разделенные символами знака равенства с двоеточием. Левая часть – это разделенные запятыми определяемые имена, после каждого из которых может явно указываться его «: ‹тип›». Правая часть состоит из списка выражений, сумма размерностей которых равна числу имен левой части определения. Выражения правой части определения не могут зависеть от имен его левой части.

Далее приведен пример *let* выражения:

```
let average := (a + b + c) / 3;
    diff_a  := (a - average) ** 2;
    diff_b  := (b - average) ** 2;
    diff_c  := (c - average) ** 2
in average, (diff_a + diff_b + diff_c)/2
end let
```

В этом примере мы определяем четыре имени: *average*, *diff_a*, *diff_b* и *diff_c*. Эти имена используются для вычисления двух значений: среднего арифметического и дисперсии чисел *a*, *b* и *c*.

4.2. Выражение *if*

Наиболее общая форма выражения *if* следующая:

```
"if", условие, "then", выражения результата,
{ "elseif", условие, "then", выражения результата },
[ "else", выражения результата ], "end", "if"
```

У всех выражений результата размерности должны быть равны, а их типы совместимы. Булевские условия вычисляются в порядке их следования. Выражения результата, идущие за первым истинным условием, определяют результат конструкции. Если все условия ложны, то ветвь *else* определяет результат конструкции. Ветвь *else* обязательна, если хотя бы одно из выражений результата является множеством типов. Когда конструкция не содержит ветви *else* и все булевские условия ложны, то возвращаются ошибочные значения. Если встретилось ошибочное булевское условие, то конструкция возвращает ошибочные значения типов идущих следом выражений результата.

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

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

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

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

4.3. Выражение *case*

Наиболее общая форма выражения *case* следующая:

```
"case", ["tag" | "type"], управляющее выражение,
  "of", условия, "then", выражения результата,
{ "of", условия, "then", выражения результата },
[ "else", выражения результата ], "end", "case"
```

Управляющее выражение может выбирать выражения результата по значению, тегу союза или по сигнатуре типа. У всех выражений результата размерности должны быть равны, а их типы совместимы, за исключением случая выбора по сигнатуре типа, где из типов каждой размерности формируется множество типов. Значение управляющего выражения сравнивается с тестами каждой условной ветви в порядке их следования до первого совпадения. Выражения результата, идущие после совпадения, определяют

результат конструкции. Если нет ни одного совпадения, то ветвь *else* определяет результат конструкции. Ветвь *else* обязательна, если хотя бы одно из выражений результата является множеством типов или типы соответствующих размерностей выражений результатов в конструкции *case type* не идентичны.

Когда конструкция не содержит ветви *else* и не произошло ни одного совпадения, то возвращаются ошибочные значения. Если управляющее выражение или встретившийся тест ошибочны, тогда возвращаются ошибочные значения типов значений идущих следом выражений результата. Если тип управляющего выражения имеет операции <= и >=, то тест может быть отрезком: «‹минимум› ! ‹максимум›». Управляющее выражение конструкции *case tag* должно быть союзом, а тесты – именами тегов этого союза. Список условий конструкции *case type* является списком типов.

Ниже приведен пример выражения *case*:

```
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
```

4.4. Выражение *where*

Выражение *where* имеет следующую форму:

```
"where", массив A, "is", имя I,
"in", выражение R, "end", "where"
```

Выражение задаёт массив такого же типа и размерностей, что и у задающего массив выражения *A*. Каждый элемент конструируемого массива равен результату выражения *R*, где может использоваться идентификатор *I*, соответствующий элементу исходного массива.

В этом примере строится массив знаков исходного массива:

```
type array_2d = array[array[integer]]
function abs(M:array_2d returns array_2d)
  where M is m in sgn(m) end where
end function
```

5. ЦИКЛИЧЕСКИЕ ВЫРАЖЕНИЯ

Язык в явном виде содержит параллельные и последовательные циклические выражения, позволяющие задавать итерационные алгоритмы для SIMD- и MIMD-архитектур. Параллельная форма детерминирована, ввиду отсутствия тупиков и конфликтов (race conditions). Последовательная форма допускает циклические зависимости, сохраняя однократность присваивания.

Циклы задаются циклической конструкцией следующего вида: «*for* ‹предложение цикла› *returns* ‹предложение возврата› *end for*». Предложение цикла содержит управление и тело цикла. Управление цикла является генератором диапазона или булевским условием для последовательных циклов. Тело цикла задает новую область со списком определений имен. Телу цикла всегда предшествует ключевое слово *repeat*, вместе с которым оно может быть опущено. Язык имеет четыре вида циклов, различающихся первыми ключевыми словами и имеющих аналогичные тела и предложения возврата. Последовательные формы циклической конструкции могут содержать блок инициализации, эквивалентный охватывающему эту циклическую конструкцию выражению *let*. Имена, используемые циклической конструкцией, делятся на три класса.

1. *Константные имена*, определяемые во внешней области действия или в блоке инициализации цикла и не определяемые его телом.
2. *Циклические имена*, определяемые телом цикла.
   - 2.1. *Циклически зависимые имена*, встречающиеся в циклической конструкции после ключевого слова *old*.
   - 2.2. *Циклически локальные имена* – это остальные циклические имена.
3. *Диапазонные имена*, определяемые в генераторе диапазона.

Значением *old* имени является значение этого имени на предыдущей итерации. На первой итерации используется обязательно заданное определение внешней области или блока инициализации. Переопределение значения имени на текущей итерации значение *old* имени не меняет.

5.1. Выражения цикла с условием

Выражение *for while* имеет следующий вид:

```
"for", [ "initial", инициализация ],
"while", условие, [ "repeat", тело],
"returns", предложение возврата, "end", "for"
```

Тело цикла выполняется, пока истинно предусловие *while*. Ключевое слово *old* может стоять перед любым циклическим именем в теле цикла и предложении возврата. Если при первом вычислении булевское выражение ложно, то ни один экземпляр тела цикла не будет вычислен.

Тело цикла выражения *for repeat* выполняется, пока истинно постусловие *while*, поэтому первый экземпляр тела цикла выполняется всегда, а циклические *old* имена могут находиться в булевском условии:

```
"for", [ "initial", блок инициализации ],
"repeat" тело цикла, "while", условие,
"returns", предложение возврата, "end", "for"
```

Далее приведем примеры циклов с условием.

Пример 1. Цикл с постусловием, вычисляющий длину строкового представления числа в десятичной форме:

```
function length(N: integer returns integer)
  for initial L := if N >= 0 then 0 else 1 end if
  repeat N := old N / 10;
         L := old L + 1
  while N ~= 0
  returns value of L
  end for
end function
```

Пример 2. Цикл с предусловием, вычисляющий математическое ожидание потока чисел:

```
function mean(S: stream[real] returns real)
  for while ~empty(S) repeat S := rest(old S)
  returns sum of first(old S)
  end for
end function
```

5.2. Циклическое выражение с диапазоном

Управляемый диапазоном цикл имеет единый вид и последователен, если содержит циклически зависимые имена, и параллелен иначе:

```
"for", генератор диапазона, [ "repeat", тело цикла ],
"returns", предложение возврата, "end", "for"
```

5.2.1. Одномерные диапазоны

Генератор одномерного диапазона определяет рассматриваемые по порядку элементы, для каждого из которых вычисляется экземпляр тела цикла. Одномерный диапазон задается триплетом или перечислением элементов массива или потока.

Триплет выглядит как: «‹имя› *in* ‹начало›, ‹конец›, ‹шаг›», где начало, конец и шаг являются унарными выражениями целого типа. Шаг диапазона необязателен и по умолчанию равен sgn(‹конец› - ‹начало›). Мощность диапазона задается формулой max(0, (‹конец› - ‹начало› + ‹шаг›) / ‹шаг›). Если мощность равна нулю, то экземпляры тела цикла не вычисляются.

Диапазон перечисления элементов массива или потока выглядит как: «имя› *in* ‹массив или поток› *at* ‹имена индексов›». Часть *at* является необязательной, но если она присутствует, то индексы перечисляемых элементов массива или потока доступны через указанные имена в соответствующих им экземплярах тела цикла. Мощность списка имен индексов *N* определяет число рассматриваемых внешних измерений массива или потока. Если число измерений массива или потока меньше *N*, то компилятор сгенерирует семантическую ошибку. Если *N* больше единицы, то диапазон является многомерным и рассматривается в разд. 5.2.2.

Два и более одномерных диапазона могут быть скомбинированы оператором *dot*, соединяющим в пары каждый элемент своего левого операнда с каждым элементом своего правого операнда, генерируя простой диапазон кортежей. Если мощности диапазонов не совпадают, тогда диапазон с меньшей мощностью дополняется ошибочными значениями.

5.2.2. Многомерные диапазоны

Существуют два вида многомерных диапазонов: диапазон массива или потока с предложением *at*, содержащим более одного имени, и цепочка одномерных диапазонов, разделенных оператором *cross*. Циклическая конструкция с многомерным диапазоном, как показано далее, выражается через циклические конструкции с одномерными диапазонами. Рассмотрим многомерное выражение размерности n многомерного цикла с *m* размерностями, где каждое предложение редукции соответствует одной размерности выражения цикла:

for D_1 **cross** D_2 **repeat** B
returns RN_1 **of** RV_1**; …;** RN_n **of** RV_n **end for**

Имя D_1 обозначает часть генератора диапазона без оператора *cross* и многомерных индексов *at*, имя D_2 соответствует оставшейся части генера-

тора диапазона, имя RN_i равно имени редукции с возможными начальными параметрами, а имя RV_i задает циклические параметры редукций. Тогда выражение многомерного цикла можно выразить через два выражения цикла с размерностями 1 и m-1:

```
for D1 repeat x1, …, xn :=
  for D2 repeat B
  returns RN11 of RV1; …; RN1n of RVn end for
returns RN21 of x1; …; RN2n of xn end for
```

Имя x_i обозначает любое уникальное имя, а имена $RN1_i$ и $RN2_i$ определяются через имя RN_i, как показано в таблице 11.

Таблица 11

Правила разложения редукций многомерных циклов

Условие на значение имени RN_i		**$RN1_i$**	**$RN2_i$**
Равно «value», «product», «least», «greatest», «catenate», «catenate (…)» или пользовательской редукции.		RN_i	value
Имеет вид «*array* [k] (i_1, …, i_k)», где: • часть «[k]» необязательна и равна «[m]» по умолчанию; • последние индексы, равные 1 по умолчанию, части «(i_1, …, i_k)» необязательны, как и вся часть.	k>1	«*array* [k_1] (i_2, …, i_k)», где $k_1 = k$-1	*array* (i_1)
	k=1	«*array* [1] (i_2, …, i_k)»	catenate (i_1)
Имеет вид «*stream* [k]», где часть «[k]» необязательна и равна «[m]» по умолчанию.	k>1	«*stream* [k_1]», где $k_1 = k$-1	*stream*
	k=1	«*stream* [1]»	catenate

Если имя D_1 содержит многомерные индексы «*n in … at* j_1, …», то многомерный цикл представляется в виде:

```
for D3 n in D4 at j1, D5 repeat B
returns RN1 of RV1; …; RNn of RVn end for
```

Имя D_3 обозначает часть генератора диапазона без операторов *cross* и многомерных индексов *at*, имя D_4 задает источник набора многомерных индексов, а имя D_5 соответствует оставшейся части генератора диапазона. Тогда выражение многомерного цикла также можно выразить через два выражения цикла с размерностями 1 и m-1:

```
for D3 n1 in D4 at j1 repeat x1, …, xn :=
  for n in n1 at D5 repeat B
  returns RN11 of RV1; …; RN1n of RVn end for
returns RN21 of x1; …; RN2n of xn end for
```

Имя n_1 обозначает любое уникальное имя, а имена $RN1_i$ и $RN2_i$ определяются через имя RN_i, как показано в таблице 11.

В качестве примера возьмем цикл с генератором диапазона, вычисляющий произведение двух матриц:

```
function multiply(A, B: array_2d returns array_2d)
  for a in A cross n in 1, size(B[1]) returns array of
    for c in a * B[1!size(B), n]
    returns sum of c end for
  end for
end function
```

5.3. Предложение возврата

Предложение возврата состоит из разделенных точкой с запятой операторов встроенных (разд. 5.3.1) и пользовательских (разд. 5.3.2) редукций, работающих над *редуцируемыми* значениями, определяемыми экземплярами тел цикла, и преобразующих их в скалярное или агрегатное значения. После оператора редукции допустимо использование фильтра вида «*when* ‹булевское выражение›», истинного для включаемых в редуцируемую цепочку значений. Порядок порождения редуцируемых значений не должен нарушать детерминированную семантику цикла, поэтому порядок вычисления тел цикла можно игнорировать только для математически ассоциативных и коммутативных редукций.

5.3.1. Встроенные редукции

Язык содержит несколько приведенных ниже встроенных редукций. Если выражения *for* и *for while* ни разу не выполняют тело цикла, то их редукции возвращают значения по умолчанию.

- Редукция «value *of x*» возвращает последнее значение *x*. По умолчанию при вычислении значения *x*, циклически зависимые *old* и не *old* имена равны значениям блока инициации или внешней области цикла, а циклически локальные имена – ошибочным значениям.
- Редукция «sum *of x*» возвращает сумму значений *x*. По умолчанию редукция равна нулю типа «*type* (*x*)».

- Редукция «product *of x*» возвращает произведение значений *x*. По умолчанию редукция равна единице типа «*type* (*x*)».
- Редукция «least *of x*» возвращает минимальное значение *x*. По умолчанию редукция равна максимальному числу типа «*type* (*x*)».
- Редукция «greatest *of x*» возвращает максимальное значение *x*. По умолчанию редукция равна минимальному числу типа «*type* (*x*)».
- Редукция «*array of x*» возвращает массив, построенный из значений *x*. По умолчанию полагается пустой массив типа «*array* [*type* (*x*)]».
- Редукция «*stream of x*» возвращает поток, построенный из значений *x*. По умолчанию полагается пустой поток типа «*stream* [*type* (*x*)]».
- Редукция «catenate *of x*» возвращает массив или поток, построенный конкатенацией значений *x*, являющихся массивами или потоками. По умолчанию редукция равна пустому массиву или потоку.
- Редукция «catenate (n) *of x*» возвращает массив с нижней границей равной n, построенный конкатенацией массивов *x*. По умолчанию редукция равна пустому массиву типа «*array* [*type* (*x*)]».

Редукции для циклической конструкции с многомерным диапазоном описываются через редукции для циклической конструкции с одномерным диапазоном, как описано в разд. 5.2.2.

5.3.2. Пользовательские редукции

Пользовательские редукции – это функции специального вида, вызываемые предложением возврата: «‹имя модуля› :: ‹имя редукции› (‹начальные значения›) *of* ‹потоковые значения›», где имя модуля (вместе с двойным двоеточием) и начальные значения вместе с окружающими их круглыми скобками можно опустить. Неоднозначность имени редукции устраняется типами начальных и потоковых значений. Общий вид определения редукции следующий:

```
"reduction", имя, "(",
  [ имена и типы начальных значений ],
  "repeat", потоковые имена и типы,
  "returns", типы результатов,
")", "for", [ "initial", инициализация ],
  "repeat", тело, "returns", предложение возврата,
"end", "for", "end", "reduction"
```

Заголовок редукции содержит имя редукции, ноль или более инициирующих значений и их типы, одно или более потоковых значений и их типы и один или более типов результата. Необязательный блок инициализа-

ции содержит список определений имен, зависящих только от начальных значений редукции. В случае конфликта имена определений блока инициализации перекрывают имена начальных значений.

Тело редукции вычисляется один раз для каждого множества потоковых значений, сгенерированных телом циклической конструкции при вызове редукции. Имя с префиксом *old* может стоять в теле и предложении возврата редукции, если это имя определяется в теле редукции, одновременно являясь начальным именем или именем блока инициализации. Предложение возврата может содержать только редукцию «value *of*».

Например, определяем редукцию, возвращающую сумму:

```
reduction sum(repeat n: num returns num)
  for initial s := 0
  repeat s := old s + n
  returns value of s
  end for
end reduction
```

СПИСОК ЛИТЕРАТУРЫА

1. **McGraw J. R.** Sisal: Streams and iterations in a single assignment language, Language Reference Manual, Version 1.2. / McGraw J. R., Skedzielewski S. K., Allan S. J., Oldehoeft R. R., Glauert J., Kirkham C., Noyce B. and Thomas R. — Livermore, CA, 1985. — (Tech. Rep. / Lawrence Livermore National Laboratory; M-146, Rev. 1).
2. **Cann D. C.** Retire Fortran?: a debate rekindled // Communications of the ACM. — New York: ACM Press, 1992. — Vol. 35, No. 8. — P. 81-89.
3. **ISO/IEC 1539-1:2004(E).** Information technology: Programming languages: Fortran: Part 1: Base language. — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 2004.
4. **ISO 7185:1990(E).** Information technology: Programming languages: Pascal. — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 1990.
5. **Böhm A. P. W.** The Sisal 2.0 reference manual / Böhm A. P. W., Cann D. C., Feo J. T. and Oldehoeft R. R. — Livermore, CA, 1991. — 128 p. — (Tech. Rep. / Lawrence Livermore National Laboratory; UCRL-MA-109098).
6. **Feo J. T.** Sisal 90 user's guide / Feo J. T., Miller P. J., Skedzielewski S. K. and Denton S. M. — Livermore, CA: Lawrence Livermore National Laboratory, Draft 0.96, 1995. — 80 p.
7. **Бирюкова Ю. В.** Sisal 90: Руководство для пользователя. — Новосибирск, 2000. — 84 с. — (Препр. / РАН. Сиб. Отд-е. ИСИ; № 72).

8. **Касьянов В. Н., Бирюкова Ю. В. и Евстигнеев В. А.** Функциональный язык Sisal // Поддержка супервычислений и интернет-ориентированные технологии. — Новосибирск, 2001. — С. 54-67.
9. **Стасенко А. П.** Транслирующие компоненты системы функционального программирования SFP / Глуханков М. П., Дортман П. А., Павлов А. А. и Стасенко А. П. // Современные проблемы конструирования программ. — Новосибирск, 2002. — С. 69-87.
10. The Unicode Standard / The Unicode Consortium. — Version 4.0. — Boston, MA: Addison-Wesley, 2003.
11. **ISO/IEC 10646:2003(E).** Information technology: Universal Multiple-Octet Coded Character Set (UCS). — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 2003.
12. **ANSI X3.4:1986.** Information systems: coded character sets: 7-Bit American national Standard Code for Information Interchange (7-Bit ASCII). — New York: American National Standards Institute (ANSI), 1986.
13. **ISO/IEC 646:1991(E).** Information technology: ISO 7-bit coded character set for information interchange. — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 1991.
14. **Робинсон У.** C# без лишних слов / Пер. с англ. — М.: ДМК Пресс, 2002. — 352 с.: ил. (Серия «Для программистов»).
15. **ISO/IEC 9899:1999(E).** Programming languages: C. — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 1999.
16. **ISO/IEC 14882:2003(E).** Programming languages: C++. — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 2003.
17. **ISO/IEC 14977:1996(E).** Information technology: Syntactic metalanguage: Extended BNF. — Geneva: Internat. Organization for Standardization (ISO), Central Secretariat, 1996.
18. **ANSI/IEEE 754-1985.** IEEE standard for binary floating-point arithmetic. — New York: Institute of Electrical and Electronics Engineers, 1985 (Reprinted in SIGPLAN Notices, 22(2):9-25, 1987).

ПРИЛОЖЕНИЕ 1. EXTENDED BNF SISAL 3.1

Каждое правило грамматики в нотации ISO Extended BNF [17] задается в виде «нетерминал = список определений | … | список определений ;». Список разделяемых запятыми определений может содержать «синтаксический множитель» или «синтаксический множитель - синтаксическое исключение». Синтаксическое исключение задает регулярное множество цепочек, исключаемых из множества цепочек, задаваемых синтаксическим множителем. Синтаксический множитель – это «синтаксическое слагаемое» или «число * синтаксическое слагаемое», где число задает точное количество повторений синтаксического слагаемого. Синтаксическое слагаемое может быть задано следующим образом:

- терминалы, исключая одиночную кавычку, в одиночных кавычках;
- терминалы, исключая двойную кавычку, в двойных кавычках;
- нетерминальные символы, имена которых могут содержать пробелы;
- сгруппированная последовательность «(список определений)» задает любое определение списка;
- опциональная последовательность «[список определений]» задает любое определение списка или пустую цепочку;
- повторяемая последовательность «{ список определений }» задает возможно пустую последовательность любых определений списка;
- текст «? любая последовательность символов, не содержащая знака вопроса ?» обрабатывается средствами, не описанными в стандарте.

Нотацией ISO Extended BNF предусмотрены комментарии вида «(* текст комментария, не содержащий последовательности символов звездочки и закрывающей круглой скобки *)».

Лексический анализ

Далее приведен синтаксис лексики языка Sisal 3.1, использующийся для преобразования потока символов программы в поток лексем.

```
(* Лексика языка Sisal 3.1 *)

(* Классы символов *)

буква =
'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' |
'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' |
```

```
'w' | 'x' | 'y' | 'z' |
'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' |
'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' |
'W' | 'X' | 'Y' | 'Z';

цифра = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

символ = ? любой символ уникода (Unicode-16) ?;

символ EOL = ? символ перевода строки с десятичным ASCII кодом
равным 10 ?;

пробельный символ = ' ' |
  ? символ табуляции с десятичным ASCII кодом равным 9 ? |
  ? символ вертикальной табуляции с десятичным ASCII кодом 11 ? |
  ? символ новой страницы с десятичным ASCII кодом равным 12 ? |
  ? символ возврата каретки с десятичным ASCII кодом равным 13 ?;

(* Лексемы *)

ключевое слово = "array" | "at" | "case" | "cross" | "declaration" |
  "definition" | "dot" | "else" | "elseif" | "end" | "error" |
  "false" | "for" | "forward" | "function" | "if" | "in" |
  "initial" | "is" | "let" | "nil" | "of" | "old" | "operation" |
  "record" | "reduction" | "repeat" | "replace" | "returns" |
  "set" | "stream" | "tag" | "then" | "true" | "type" | "union" |
  "uses" | "when" | "where" | "while";

идентификатор = ((буква | '_'), {буква | цифра | '_'}) -
                (ключевое слово | '_');
esc-символ = "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v" |
             "\\" | '\"' | "\'";

esc-код = '\', (цифра, {цифра} | 'o', цифра, {цифра} |
          'h', (цифра | буква), {цифра | буква});

символьный литерал = "'", (символ - ('\' | символ EOL) |
  esc-символ | esc-код), "'";

обычный строковой литерал = '"', { esc-код, {esc-код}, ';' |
  символ - ('\' | '"' | символ EOL) | esc-символ }, '"';

буквальный строковой литерал = "@"", {
  символ - ('\' | '"' | символ EOL) |
  '\', символ - ('"' | символ EOL) | "\""}, '"';

строковой литерал =
  обычный строковой литерал | буквальный строковой литерал;

склеенный строковой литерал = строковой литерал,
  {{пробельный символ | символ EOL}, строковой литерал};

много цифр = цифра, {цифра | '_'};

много цифр с точкой = ([много цифр], '.', [много цифр]) - '.';

много цифр со знаком = {'_'}, ['-' | '+'], {'_'}, много цифр;

натуральный литерал = много цифр |
  много цифр, '#', {'_'}, (цифра | буква), {цифра | буква | '_'};
```

```
вещественный литерал одинарной точности =
  много цифр, ('e' | 'E'), много цифр со знаком |
  много цифр с точкой, [('e' | 'E'), много цифр со знаком];

вещественный литерал двойной точности = (много цифр |
  много цифр с точкой), ('d' | 'D'), [много цифр со знаком];

(* Одно-символьные лексемы '*' ':' '<' '>' '|' и '~'
   выделяются только если они не всходят в состав двух-символьных *)
лексема = '!' | '%' | '&' | '(' | ')' | '*' | '+' | ',' | '-' |
  '.' | '/' | ':' | ';' | '<' | '=' | '>' | '[' | ']' | '^' | '{' |
  '|' | '}' | '!' | '~' | "**" | "::" | ":=" | "<=" | ">=" | "||" |
  "~=" | ключевое слово | идентификатор | '#', идентификатор |
  символьный литерал | склеенный строковой литерал |
  натуральный литерал | вещественный литерал одинарной точности |
  вещественный литерал двойной точности;

(* Комментарий *)

строчный комментарий = "//", {символ - символ EOL}, символ EOL;

блочный комментарий =
  "/*", {символ} - ({символ}, "*/", {символ}), "*/";

(* Директивы препроцессора *)

начало директивы = {пробельный символ}, '#'  ;

строка директивы = {пробельный символ | блочный комментарий},
  (строчный комментарий | символ EOL);

строки директивы = пробельный символ, лексическая свертка строки,
  {{пробельный символ}, '#', пробельный символ,
   лексическая свертка строки};

пробельные символы = пробельный символ, {пробельный символ};

директива препроцессора = директива if | начало директивы, (
  "undef", пробельные символы, идентификатор, строка директивы |
  ("define" | "line" | "error" | "warning"), строки директивы |
  "region", строки директивы, лексическая свертка,
  начало директивы, "endregion", строки директивы);

(* Директива условной трансляции *)

не знак директивы =
  символ - (пробельный символ, '#'), {символ - символ EOL};

есть комментарий = {символ}, ("/*" | "//"), {символ};

не директива препроцессора = {пробельный символ},
  [(не знак директивы - есть комментарий | блочный комментарий),
  {{символ - символ EOL} - есть комментарий | блочный комментарий}],
  (строчный комментарий | символ EOL);

пропускаемый текст =
  {директива препроцессора | не директива препроцессора};

(* Выбор контролируется условной директивой *)
секция текста = лексическая свертка | пропускаемый текст;

директива if =
   начало директивы, "if", строки директивы, секция текста,
```

```
  {начало директивы, "elseif", строки директивы, секция текста},
  [начало директивы, "else", строка директивы, секция текста],
   начало директивы, "endif", строка директивы;

(* Лексическая свертка *)

не лексема = {пробельный символ | блочный комментарий};

лексическая свертка строки = {не лексема}, {лексема, {не лексема}},
  (строчный комментарий, символ EOL);

лексическая свертка =
  {лексическая свертка строки | директива препроцессора};
```

Синтаксический анализ

Далее приведен синтаксис языка Sisal 3.1, использующийся при преобразовании потока лексем (каждая последовательность терминалов соответствует текстовому представлению лексемы) в синтаксическое дерево.

```
(* Синтаксис языка Sisal 3.1 *)

(* Модуль *)

интерфейс модуля = "declaration", идентификатор,
  ["uses", список идентификаторов],
  {определение типа | объявление модуля};

реализация модуля = "definition", идентификатор,
  ["uses", список идентификаторов], {определения модуля};

объявление модуля = объявление функции | объявление операции |
                    объявление редукции;

определение модуля = определение типа | определение функции |
                     определение операции | определение редукции |
                     "forward", объявление модуля;

(* Имена *)

имя объекта модуля = [идентификатор, "::"], идентификатор;

список идентификаторов = идентификатор, {',', идентификатор};

(* Определение типа *)

определение типа = определение переименования типа |
                   определение пользовательского типа;

определение переименования типа = идентификатор, '=', тип;

определение пользовательского типа = идентификатор, ":=", тип;

тип = имя объекта модуля |
  "type", '(', выражение, ')' |
  "array", '[', тип, ']' |
  "stream", '[', тип, ']' |
  "record", '[', список полей, ']' |
  "union", '[', список тегов, ']' |
  "function", '[', [список типов], "returns", список типов, ']' |
```

```
  "set", '[', список типов, ']' |
  "array" | "stream" | "record" | "union" | "function";

список типов = тип, {',', тип};

список полей = список идентификаторов, ':', тип,
         {';', список идентификаторов, ':', тип};

список тегов = список идентификаторов, [':', тип],
         {';', список идентификаторов, [':', тип]};
(* Функция *)

заголовок определения =
  '(', [список полей], "returns", список типов, ')';

заголовок объявления = заголовок определения |
  '[', [список типов], "returns", список типов, ']';

объявление функции =
  "function", идентификатор, заголовок объявления;

определение функции = "function", идентификатор,
  заголовок определения, список выражений, "end", "function";

λ-функция = "function", заголовок определения,
            список выражений, "end", "function";

(* Операция *)

объявление операции =
  "operation", знак операции, заголовок объявления;

определение операции =
  "operation", знак операции, заголовок определения,
                       список выражений, "end", "operation"
знак операции = '%' | '&' | '*' | '+' | '-' | '.', идентификатор |
                '/' | ':' | '<' | '=' | '^' | '|' | '~' | '' |
                "**" | "||" | '(', ')' | '[', ']';
(* Редукция *)

заголовок определения редукции = '(', [список полей], "repeat",
  список полей, "returns", список типов, ')';

объявление редукции = "reduction", идентификатор,
  (заголовок определения редукции | '[', [список типов], "repeat",
   список типов, "returns", список типов, ']');

инициализация = "initial", список определений;

тело цикла = "repeat", список определений;

определение редукции =
  "reduction", идентификатор, заголовок определения редукции,
    "for", [инициализация], тело цикла,
    "returns", "value", "of", список выражений,
         {';', "value", "of", список выражений}
    "end", "for",
   "end", "reduction";

(* Выражение (с учетом ассоциативности) *)

выражение = конкатенация;
список выражений = выражение, {',', выражение};
```

```
конкатенация = конкатенация, "||", логическое OR  | логическое OR;

логическое OR = логическое OR, '|', логическое XOR | логическое XOR;

логическое XOR =
  логическое XOR, '^', логическое AND | логическое AND;

логическое AND = логическое AND, '&', сравнение | сравнение;

сравнение =
  сравнение, '=', плюс и минус | сравнение, "~=", плюс и минус |
  сравнение, '<', плюс и минус | сравнение, "<=", плюс и минус |
  сравнение, '>', плюс и минус | сравнение, ">=", плюс и минус |
  плюс и минус;

плюс и минус = плюс и минус, '+', умножить и разделить |
               плюс и минус, '-', умножить и разделить |
               умножить и разделить;

умножить и разделить = умножить и разделить, '*', степень |
                       умножить и разделить, '/', степень |
                       умножить и разделить, '%', степень |
                       степень;

степень = префиксные операции, "**", степень | префиксные операции;

префиксные операции = {'-', '+', '~'}, постфиксные операции;

постфиксные операции = ('(', выражение, ')' | операнд),
  {'(', (выражение | ':', тип), {',', (выражение | ':', тип)}, ')' |
   '[', выбор или замена элементов массива, ']' |
   "replace", '{', список определений, '}' |
   '.', идентификатор | "tag", идентификатор | ':', тип};

(* Операнд *)

операнд = имя объекта модуля | "old", идентификатор |
  "function", имя объекта модуля, ['[', список типов, ']'] |
  "operation", знак операции, '[', список типов, ']' |
  "operation", (':' | ''), '[', тип, "returns", тип, ']' |
  константа | выражение let | выражение if | выражение where |
  выражение case | выражение case tag | выражение case type |
  выражение цикла | λ-функция | массив | поток | запись | союз |
  "is", "error", '(', выражение, ')' |
  "type", '(', выражение, ')' |
  "type", '[', тип, ']' |
  "error", '[', тип, ']';

(* Константа *)

константа = "true" | "false" | "nil" | натуральный литерал |
            символьный литерал | строковой литерал |
            вещественный литерал одинарной точности |
            вещественный литерал двойной точности;

(* Выражение let *)

выражение let = "let", список определений,
                "in", список выражений, "end", "let"
```

```
список определений =
  левая часть определения, ":=", список выражений,
  {';', левая часть определения, ":=", список выражений};

левая часть определения = (идентификатор, [':', тип] |),
                    {',', (идентификатор, [':', тип] |)};
(* Выражение if *)

выражение if = "if", выражение, "then", список выражений,
               {"elseif", выражение, "then", список выражений}
               ["else", список выражений], "end", "if";

(* Выражение where *)

выражение where = "where", выражение, "is", идентификатор,
                  "in", выражение, "end", "where";

(* Выражение case *)

выражение case = "case", выражение,
  "of", условие case of, {',', условие case of},
  "then", список выражений,
  {"of", условие case of, {',', условие case of},
   "then", список выражений},
  ["else", список выражений], "end", "case";

условие case of = выражение | выражение, '!', выражение;

(* Выражение case tag *)

выражение case tag = "case", "tag", выражение,
   "of", список идентификаторов, "then", список выражений,
  {"of", список идентификаторов, "then", список выражений},
  ["else", список выражений], "end", "case";

(* Выражение case type *)

выражение case type = "case", "type", выражение,
   "of", список типов, "then", список выражений,
  {"of", список типов, "then", список выражений},
  ["else", список выражений], "end", "case";

(* Выражение цикла *)

выражение цикла = "for",
  (предложение for, предложение for while, предложение for repeat),
  "returns", вызов редукции, {';', вызов редукции}, "end", "for";

предложение for = генератор диапазона for, [тело цикла];

предложение for while =
  [инициализация], "while", выражение, [тело цикла];

предложение for repeat =
  [инициализация], тело цикла, "repeat", выражение;

генератор диапазона for = cross операции for;

cross операции for = dot операции for, {"cross", dot операции for};

dot операции for = диапазон for, {"dot", диапазон for};

диапазон for = идентификатор, "in", выражение, 2*[',', выражение],
               ["at", список идентификаторов];
```

```
вызов редукции = (имя объекта модуля |
                  array, [‘[’, натуральный литерал, ‘]’] |
                  stream, [‘[’, натуральный литерал, ‘]’]),
  [‘(’, список выражений, ‘)’], “of”, список выражений,
  [“when”, список выражений];

(* Конструкторы типов *)

массив = [“array”], ‘{’, [выражение, ‘;’], список выражений, ‘}’ |
  ([“array”], имя объекта модуля | “array”, ‘[’, тип, ‘]’),
  ‘{’, [выражение, ‘;’], [список выражений], ‘}’;

поток = “stream”, ‘{’, список выражений, ‘}’ |
  ([“stream”], имя объекта модуля | “stream”, ‘[’, тип, ‘]’),
  ‘{’, [список выражений], ‘}’;

запись = “record”, ‘{’, список определений, ’}’ |
  ([“record”], имя объекта модуля |
   “record”, ‘[’, список полей, ‘]’),
  ‘{’, (список определений | “:=”, список выражений), ’}’;

союз =
  ([“union”], имя объекта модуля | “union”, ‘[’, список тегов, ‘]’),
  ‘{’, идентификатор, “:=”, выражение, ’}’;

(* Выбор или замена элементов массива *)

выбор или замена элементов массива =
  выбор элементов массива, [“:=”, список выражений],
  {‘;’, выбор элементов массива, [“:=”, список выражений]};

выбор элементов массива = cross операции;

cross операции = dot операции, {(“cross” | ‘,’), dot операции};

dot операции = диапазон, {“dot”, диапазон};

диапазон = [идентификатор, “in”],
           ([выражение], [‘!’, [выражение], [‘!’, выражение]]) - “”;
```

ПРИЛОЖЕНИЕ 2. ПРИМЕР МОДУЛЯ, РЕАЛИЗУЮЩЕГО КОМПЛЕКСНОЕ ЧИСЛО

Интерфейс модуля

```
// Объявление комплексного типа одинарной точности.
declaration complex
uses double_complex

// Объявляется как пользовательский тип (:=).
type complex := record [ r, i: real ]

// Конструктор комплексного значения.
function complex [real, real   returns complex]

// Операции приведения типа.
operation  [real   returns complex]
operation :[double returns complex]

// Значение действительной и мнимой части комплексного
числа.
operation .r [complex      returns real]
operation .i [complex      returns real]
// Вычисляемое значение модуля и главного значения
// аргумента комплексного числа.
operation .mod [complex    returns function [returns real]]
operation .phi [complex    returns function [returns real]]

// Операция унарной идентичности и смены знака.
operation +  [complex      returns complex]
operation -  [complex      returns complex]

// Операция равенства (автоматически определяет операцию
// неравенства). Равенство двух значений типа complex
// по умолчанию генерируется автоматически.
operation =  [complex, real    returns boolean]
operation =  [complex, double  returns boolean]

// Операция сравнения «меньше» (автоматически определяет
// все другие операции сравнения)
operation <  [complex, real    returns boolean]
operation <  [complex, double  returns boolean]
```

```
operation <  [complex, complex returns boolean]
operation <  [real,    complex returns boolean]
operation <  [double,  complex returns boolean]

// Операция сложения.
operation +  [complex, real    returns complex]
operation +  [complex, double  returns double_complex]
operation +  [complex, complex returns complex]

// Операция умножения.
operation *  [complex, real    returns complex]
operation *  [complex, double  returns double_complex]
operation *  [complex, complex returns complex]

// Операция вычитания.
operation -  [complex, real    returns complex]
operation -  [complex, double  returns double_complex]
operation -  [complex, complex returns complex]
operation -  [real,    complex returns complex]
operation -  [double,  complex returns double_complex]

// Операция деления.
operation /  [complex, real    returns complex]
operation /  [complex, double  returns double_complex]
operation /  [complex, complex returns complex]
operation /  [real,    complex returns complex]
operation /  [double,  complex returns double_complex]

// Операция возведения в степень. Возвращается главное
значение.
operation ** [complex, real    returns complex]
operation ** [complex, double  returns double_complex]
operation ** [complex, complex returns complex]
operation ** [real,    complex returns complex]
operation ** [double,  complex returns double_complex]
```

Реализация модуля

```
// Определение комплексного типа одинарной точности.
definition complex
uses double_complex, math

type complex_record = record [ r, i: real ]

function complex (r, i: real returns complex)
```

```
  record complex_record {r := r; i := i}
end function

operation (r: real returns complex)
  record complex_record {r := r; i := 0}
end operation

operation :(r: double returns complex)
  record complex_record {r := r:real; i := 0}
end operation

operation .r (c: complex returns real)
  c : complex_record . r
end operation

operation .i (c: complex returns real)
  c : complex_record . i
end operation

operation .mod (c: complex returns function [returns real])
  function (returns real)
    (c.r*c.r + c.i*c.i) ** 0.5
  end function
end operation

operation .phi (c: complex returns function [returns real])
  function (returns real)
    math::atan(c.i, c.r)
  end function
end operation

operation + (c: complex returns complex)
  c
end operation

operation - (c: complex returns complex)
  complex(-c.r, -c.i)
end operation

operation = (c: complex; r: real returns boolean)
  c.r = r & c.i = 0
end operation

operation = (c: complex; d: double returns boolean)
  c.r = d & c.i = 0
end operation
```

```
operation < (c: complex; r: real returns boolean)
  c.mod() < math::abs(r)
end operation

operation < (c: complex; d: double returns boolean)
  c.mod() < math::abs(d)
end operation

operation < (c1, c2: complex returns boolean)
  c1.mod() < c2.mod()
end operation

operation < (r: real; c: complex returns boolean)
  math::abs(r) < c.mod()
end operation

operation < (d: double; c: complex returns boolean)
  math::abs(d) < c.mod()
end operation

operation + (c: complex; r: real returns complex)
  c : complex_record replace {r := c.r+r}
end operation

operation + (c: complex; d: double returns double_complex)
  double_complex(c.r+d, c.i)
end operation

operation + (c1, c2: complex returns complex)
  complex(c1.r+c2.r, c1.i+c2.i)
end operation

operation * (c: complex; r: real returns complex)
  complex(c.r*r, c.i*r)
end operation

operation * (c: complex; d: double returns double_complex)
  double_complex(c.r*d, c.i*d)
end operation

operation * (c1, c2: complex returns complex)
  complex(c1.r*c2.r - c1.i*c2.i, c1.r*c2.i + c1.i*c2.r)
end operation

operation - (c: complex; r: real returns complex)
```

```
  c : complex_record replace {r := c.r-r}
end operation

operation - (c: complex; d: double returns double_complex)
  double_complex(c.r-d, c.i)
end operation

operation - (c1, c2: complex returns complex)
  complex(c1.r-c2.r, c1.i-c2.i)
end operation

operation - (r: real; c: complex returns complex)
  complex(r-c.r, c.i)
end operation

operation - (d: double; c: complex returns double_complex)
  double_complex(d-c.r, c.i)
end operation

operation / (c: complex; r: real returns complex)
  complex(c.r/r, c.i/r)
end operation

operation / (c: complex; d: double returns double_complex)
  double_complex(c.r/d, c.i/d)
end operation

operation / (c1, c2: complex returns complex)
  let mod2 := c2.r*c2.r + c2.i*c2.i
  in complex((c1.r*c2.r + c1.i*c2.i) / mod2,
             (c1.i*c2.r - c1.r*c2.i) / mod2)
  end let
end operation

operation / (r: real; c: complex returns complex)
  let f := r / (c.r*c.r + c.i*c.i)
  in complex(c.r*f, -c.i*f)
  end let
end operation

operation / (d: double; c: complex returns double_complex)
  let f := d / (c.r*c.r + c.i*c.i)
  in double_complex(c.r*f, -c.i*f)
  end let
end operation
```

```
operation ** (c: complex; r: real returns complex)
  let mod := (c.r*c.r + c.i*c.i) ** (r / 2.0);
      phi := r * c.phi()
  in complex(mod * math::cos(phi), mod * math::sin(phi))
  end let
end operation

operation ** (c: complex; d: double returns double_complex)
  let mod := (c.r*c.r + c.i*c.i) ** (d / 2.0d);
      phi := d * c.phi()
  in double_complex(mod * math::cos(phi), mod *
math::sin(phi))
  end let
end operation

operation ** (c1, c2: complex returns complex)
  let exp_mod1 := math::log (c1.r*c1.r + c1.i*c1.i) / 2.0;
      phi1     := c1.phi();
      mod      := math::exp (c2.r*exp_mod1 - c2.i*phi1);
      phi      := c2.i*exp_mod1 + c2.r*phi1
  in complex(mod * math::cos(phi), mod * math::sin(phi))
  end let
end operation

operation ** (r: real; c: complex returns complex)
  let mod := r ** c.r;
      phi := c.i * math::log(r)
  in complex(mod * math::cos(phi), mod * math::sin(phi))
  end let
end operation

operation ** (d: double; c: complex returns double_complex)
  let mod := d ** c.r;
      phi := c.i * math::log(d)
  in double_complex(mod * math::cos(phi), mod *
math::sin(phi))
  end let
end operation
```

А. П. Стасенко, А. И. Синяков

БАЗОВЫЕ СРЕДСТВА ЯЗЫКА SISAL 3.1

Препринт
132

Рукопись поступила в редакцию 27.01.06
Редактор З. В. Скок
Рецензент В. А. Евстигнеев

Подписано в печать 25.07.06
Формат бумаги 60 × 84 1/16 Объем 3.3 уч.-изд.л., 3.6 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2», г. Бердск, 49-а, оф. 7, тел./факс 8 (241) 5 38 77