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

В. И. Шелехов

ЯЗЫК ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ Р

Препринт
101

Новосибирск 2002

Описывается язык предикатного программирования **P** (**P**redicate programming language). Предикатная программа есть рекурсивно определяемая система вычислимых логических утверждений. На языке P можно запрограммировать любое алгоритмическое решение задачи, спецификация которой представима в виде математического предиката. Методология предикатного программирования ориентирована на построение предикатной программы, из которой можно получить эффективную императивную программу применением системы оптимизирующих трансформаций.

Возможности языка P существенно шире известных языков функционального программирования. Наряду с функциональным стилем программирования применяется предикатный (операторный) стиль. Удобство и гибкость программирования обеспечивается за счет использования аппарата гиперфункций. Язык определяет развитую систему типов данных. Массивы определяются конструктором forAll (имеющим сходство с циклом forAll в FORTRAN-90) и его производными формами. Параллелизм программы естественным и явным образом определяется конструкциями параллельного оператора и конструктора forAll.

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

V. I. Shelekhov

THE PREDICATE PROGRAMMING LANGUAGE P

Preprint
101

Novosibirsk 2002

The **P**redicate programming language P is described here. A predicate program is the collection of recursive computable logical equations. It is possible to write a predicate program for each algorithmic decision of a problem whose specification may be written in the predicate form.

Facilities of the P language for describing algorithms are more powerful than those of existing functional languages. Along with the functional programming style, the predicate (statement) programming style is applicable. In a predicate program, an array is defined by the forAll constructor which is like the parallel forAll loop in the FORTRAN-90 language. Parallel algorithms may be naturally expressed by means of parallel statements and forAll constructors.

ВВЕДЕНИЕ

Описывается язык предикатного программирования **P: P**redicate programming language. Программа на языке P называется **предикатной программой** или **P-программой**. Предикатная программа есть рекурсивно определяемая система вычислимых логических утверждений. Она представлена в форме, удобной для программирования. Логические прототипы конструкций предикатной программы, принципы построения языка P и методология предикатного программирования описаны в работе [1]. Возможности языка P существенно шире известных языков функционального программирования [3–7]. Любое алгоритмическое решение задачи, спецификация которой представима в виде математического предиката, может быть оформлено в виде предикатной программы. Методология предикатного программирования ориентирована на построение предикатной программы, из которой можно получить эффективную императивную программу применением системы оптимизирующих трансформаций.

Исполняемыми конструкциями языка P являются **выражения** и **операторы.** Результатом исполнения выражения всегда является некоторое значение. Оператор является **предложением**, если результатом его вычисления может быть (или не быть) значение в зависимости от позиции, в которую входит предложение. Остальные операторы (не предложения) никогда не вырабатывают значения как результата конструкции. Использование предложений позволяет при программировании на языке P гибко комбинировать функциональный и предикатный (операторный) стили. Подобными свойствами обладает лишь язык Алгол-68 [2].

Произвольный **тип** данных строится из **примитивных** типов, которыми являются: натуральные (**nat**), целые (**int**), вещественные (**real**), логические (**bool**), символьные (**char**) и комплексные (**complex**). **Подмножество типа** определятся как область истинности некоторого предиката. Тип, совместимый с типом “целый”: **int**, **nat** и любое их подмножество — называется **интегральным** типом. **Структурными** типами являются: массивы, кортежи, объединения, последовательности и множества. Типы могут быть **параметризованы** не только компонентными типами, но и переменными, используемыми в предикатах при образовании подмножеств. Будем использовать понятие **совместимости** типов, подразумевая возможность приведения значения одного типа к другому типу при подстановке параметров и в арифметических операциях.

Для описания синтаксиса предикатной программы используется расширенный язык Бэкусовских нормальных форм (БНФ). Фрагмент синтаксического определения — последовательность терминальных и нетерминальных символов — может быть обрамлена квадратными или фигурными скобками:

[<фрагмент>] — означает возможное отсутствие <фрагмента>;

{<фрагмент>} — допускает многократное повторение <фрагмента>.

Терминальные символы {, }, [,], |, || и := будем изображать с подчеркиванием: {, }, [,] , |, || и :=.

В данной работе предлагается систематическое описание языка P. Описываемая версия языка является развитием его начальной версии, изложенной в работе [1], и отражает опыт его разработки как языка программирования для небольшого числа задач. В разд. 1 определяются набор лексем и общие правила оформления текста программы. В разд. 2 дается общее описание основной конструкции предикатной программы — определения предиката. Определяется понятие гиперфункции, которая является более общей формой предиката по сравнению с обычной функцией. В разд. 3 описывается общая структура предикатной программы. В разд. 4 описываются операторы предикатной программы. В частности, определяется предложение расщепления на базе гиперфункции, реализующее принципиально новый способ организации ветвления в программе. В разд. 5 описываются выражения; определяется типовой набор операций. В разд. 6. представлена система типов. В разд. 7 определяется конструктор массивов и его производные формы. В разд. 8 описывается императивное расширение языка, содержащее оператор присваивания, циклы **loop**, **while** и **for** и другие конструкции, появляющиеся в программе в результате применения трансформаций. В разд. 9 дается дополнительная семантика языка P, определяющая потоковые свойства программы. Заключение содержит замечания по описываемой версии языка и дальнейшие планы его развития.

1. ТЕКСТ ПРОГРАММЫ. ЛЕКСЕМЫ

Текст программы хранится в файлах, называемых единицами компиляции. Использование определений из другой единицы компиляции реализуется через директиву #include. Текст единицы компиляции представлен в виде последовательности строк символов. Переход на новую строку эквивалентен символу “пробел”, изображаемому далее в описании синтаксиса нетерминальным символом <пробел>.

Текст программы состоит из последовательности символов за исключением комментариев. Комментарий начинается парой символов /* и завершается парой символов */. Второй вид комментариев начинается с пары символов // и продолжается до конца текущей строки файла.

Текст программы составляется из лексем следующего вида:

<лексема> ::= <имя> | <изображение константы> | <ограничитель>
<ограничитель> ::= <разделитель> | <операция> | <скобка>|
<кавычка> | <служебное слово>
<разделитель> ::= # | : | ; | . | ? | , | ! | $ | @ | -> | <пробел>
<скобка> ::= { | } | [|] | (|)
<операция> ::= < | > | % | * | - | + | / | ^ | & | _ | ~ | ! | = | \
<кавычка> ::= ' | "

Между двумя лексемами может находиться любое число пробелов, которые считаются незначащими и могут быть удалены из текста программы. Исключение составляют случаи, когда пробел используется в качестве разделителя в синтаксической конструкции.

<имя> ::= <буква> [<продолжение имени>]
<продолжение имени> ::=
<буква> [<продолжение имени>] |
<цифра> [<продолжение имени>]
<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| _
<восьмеричная цифра> ::= 0| 1| 2| 3| 4| 5| 6| 7
<цифра> ::= <восьмеричная цифра> | 8 | 9
<шестнадцатеричная цифра> ::= <цифра> | A | B | C | D | E | F
<изображение константы> ::=
<изображение натуральной константы> |
<изображение целой константы> |
<изображение вещественной константы> |
<изображение символьной константы> |
<изображение строковой константы>
<изображение натуральной константы> ::=
<цифра> [<изображение натуральной константы>]
<натуральное> ::= <изображение натуральной константы>
<изображение целой константы> ::=
[<знак числа>] <натуральное>
<знак числа> ::= + | -
<изображение вещественной константы> ::=
[<знак числа>] [<натуральное>.]<натуральное> [<порядок>] |

[<знак числа>] .<натуральное> [<порядок>] |
[<знак числа>] <натуральное>. [<порядок>]
<порядок> ::= e[<знак числа>]<натуральное> |
E[<знак числа>]<натуральное>
<изображение символьной константы> ::= '<литера>'
<литера> ::= <буква> | <цифра> | <разделитель> | <операция> | <скобка> |
<специальное изображение литеры>
<специальное изображение литеры> ::=
\' | \" | \? | \\ | \a | \b | \f | \n | \r | \t | \v |
\<изображение восьмеричной константы> |
\x<изображение шестнадцатеричной константы>
<изображение восьмеричной константы> ::=
<восьмеричная цифра> [<изображение восьмеричной константы>]
<изображение шестнадцатеричной константы> ::=
<шестнадцатеричная цифра>
[<изображение шестнадцатеричной константы>]

Специальное изображение литеры используется для кавычек и других специальных символов по правилам языков C и C++.

<изображение строковой константы> ::= "<тело строковой константы>"
<тело строковой константы> ::= <литера> [<тело строковой константы>]
<служебное слово> ::=
array | **bool** | **case** | **char** | **complex** | **do** | **else** | **elscase** |
elsCase | **elsif** | **end** | **exit** | **false** | **fin** | **for** | **forall** |
forAll | **forcase** | **forCase** | **global** | **if** | **in** | **int** |
loop | **lambda** | **nat** | **of** | **or** | **predicate** | **real** |**return** |
seq | **set** | **split** | **step** | **string** | **struct** | **then** | **true** |
type | **union** | **while**

Служебные слова не могут быть использованы в качестве имен программы. Следующие пары служебных слов эквивалентны: **elscase** и **elsCase**, **forall** и **forAll**, **forcase** и **forCase.**

2. ПРЕДИКАТЫ

Предикатом является условие, определяющее зависимость между значениями переменных — **параметров** предиката. Это условие обычно выразимо в виде формулы на языке исчисления предикатов.

Основной конструкцией языка P является **определение предиката**. Параметры предиката подразделяются на две части: **исходные** (или входные)

и **результирующие** (или неизвестные). **Вычисление предиката** реализует процесс получения значений результирующих параметров по значениям исходных. Естественно, что предикат должен быть истинным на всей совокупности значений параметров.

Будем использовать запись A(x**:** z), где A обозначает имя предиката, x — набор исходных переменных $x_1,x_2,\ldots,x_n$ ($n \geq 0$), а z — набор результирующих переменных $z_1,z_2,\ldots,z_m$ ($m>0$); наборы x и z не пересекаются, т.е. любая переменная может присутствовать только в одном из наборов. В случае, когда требуется вычислить логическое значение предиката A (набор z пуст), будем использовать в качестве результата дополнительный параметр логического типа, например b, и записывать предикат в виде A(x: b).

Определение предиката в языке P имеет следующую форму:

```
<определение предиката> ::=
                <заголовок предиката> <тело предиката> fin [;]
<тело предиката> ::= <оператор>
```

Вхождение некоторого предиката B(x**: z**) в теле другого предиката называется **вызовом** предиката. Вхождение предиката в заголовке предиката называется **определяющим**. Вычисление предиката реализуется исполнением тела предиката.

В языке P имеются также **примитивные предикаты** со стандартными зарезервированными именами. Примитивный предикат не имеет тела.

```
<заголовок предиката>::=
     <имя предиката>(<описание исходных и результирующих параметров>)
<имя предиката >::=<имя>
<описание исходных и результирующих параметров>::=
     [<описания или обозначения параметров>]:
     <описания или обозначения результатов>
<описания или обозначения параметров>::=
     <описания параметров> | <обозначения параметров>
<описания параметров>::=
     <изображение типа><пробел><имя параметра>[*]
      [{,<имя параметра>[*]}] [,<описания параметров>]
<имя параметра>::=<имя>
```

<изображение типа> декларирует тип одного или нескольких следующих за ним параметров (см. разд. 6); <пробел> есть терминальный символ (см. разд. 1). **Тип** определяет множество значений, допустимых для каждого параметра.

Наличие ограничителя * после <имени параметра> подразумевает, что конструкция <имя параметра>* наряду с входным параметром описывает также результирующий параметр с именем <имя параметра>’ и тем же типом, что и у входного параметра. В императивной программе, реализуемой для предикатной программы, результирующий параметр склеивается с соответствующим входным параметром. Параметр с именем <имя параметра>’ не должен упоминаться в описании результирующих параметров для данного определения предиката. Он считается находящимся в начале списка результирующих параметров первой ветви (см. ниже).

<обозначения параметров>::=
<имя параметра>[*] [,<обозначения параметров>]

В данном случае типы параметров не декларируются. Определение типов параметров возлагается на транслятор с языка Р.

Пример 2.1. Определение предиката sign для знака вещественного числа x:

```
sign(real x: int s)
        if x>0 then s = 1 elsif x = 0 then s = 0 else s = -1 end
fin
```

В определении предиката sign параметр x является входным, результирующий параметр s есть значение знака числа x. Условное предложение в теле предиката sign определяет s в зависимости от условий x>0, x = 0 и x>0.

Для вызова предиката A(x: z) допускается также **функциональная форма** записи. A(x) обозначает значение результирующего набора “z” для исходного набора “x”; т.е. A(x: z) эквивалентно z = A(x). Конструкцию A(x) будем называть **вызовом функции**. Для предиката A(x: b), где b — логическая переменная, используется функциональная форма A(x) в позиции, где требуется логическое значение предиката.

Возможны две разные формы для тела предиката: **предикатная** (операторная) и **функциональная**. В предикатной форме результирующие параметры должны быть определены в теле явным образом через предикат равенства, вызовы предикатов или другим образом. В функциональной форме тело предиката имеет форму предложения и его исполнение завершается вычислением выражения для единственного результирующего параметра или списка выражений при наличии нескольких результирующих параметров. Типы выражений должны соответствовать типам соответствующих результирующих параметров.

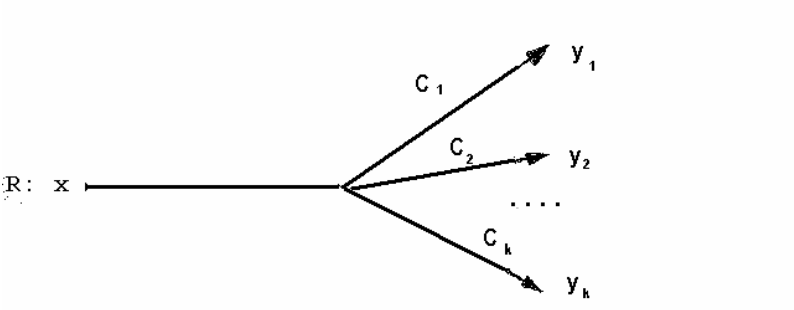
Пример 2.2. Определение предиката sign в функциональной форме:

```
sign(real x: int)
        if x>0 then 1 elsif x = 0 then 0 else -1 end
fin
```

Имя результирующего параметра s опускается в заголовке предиката sign.

Предикат вида A(x: z) имеет форму функции, отображающей значения переменных набора x в значения переменных набора z. Возможна более общая форма предиката в виде **гиперфункции** $R(x: y_1| y_2| \ldots| y_k)$ с k **ветвями**, где R — имя предиката, $k > 0$, $y_1, y_2, \ldots, y_k$ — попарно непересекающиеся, возможно пустые, наборы переменных. С гиперфункцией R связан также полный набор взаимно исключающих условий истинности ветвей: набор предикатов $C_1(x), C_2(x),\ldots,C_k(x)$ таких, что $C_1(x)\vee C_2(x)\vee\ldots\vee C_k(x)$ и $(\forall i,j=1..k)(i\neq j \Rightarrow \neg(C_i(x)\&C_j(x)))$ — тождественно истинные формулы. В случае истинности условия $C_i(x)$ предикат R связывает набор x со значениями переменных набора y_i, при этом значения переменных для других наборов y_j $(i \neq j)$ не определены.

Графически гиперфункция изображается следующим образом:



В случае $k = 1$ гиперфункция вырождается в обычную функцию. При $k > 1$ гиперфункция представляет собой гибрид функции и распознавателя; ее вычисление реализует ветвление.

Исполнение гиперфункции завершается некоторой ветвью i с вычислением значений переменных набора y_i. Отметим, что выбор ветви реализуется внутри вычисления гиперфункции, тогда как при исполнении условного предложения **if** D **then** A **else** B **end** ветвление реализуется после вычисления условия D.

Наконец, представим правила описания результирующих параметров в заголовке предиката.

<описания или обозначения результатов>::=
[<описания ветви результатов>]
[⊥ <описания или обозначения результатов>]

После разделителя "|" определяются параметры результирующего набора переменных для следующей ветви гиперфункции. Ветвь с пустым результирующим набором является **пустой**.

<описания ветви результатов> ::=
<описания или обозначения результирующих параметров> |
<описания типов результирующих параметров>
<описания или обозначения результирующих параметров> ::=
<описания результирующих параметров> |
<обозначения результирующих параметров>
<описания результирующих параметров>::=
<изображение типа><пробел><имя результата>[{,<имя результата>}]
[,<описания результирующих параметров>]
<имя результата> ::= <имя> | <имя склеиваемого результата>
<имя склеиваемого результата> ::= <имя>'

Имя вида <имя>' используется для именования результирующего параметра предиката, при условии что <имя> описано в качестве входного параметра с тем же типом. В императивной программе, реализуемой для предикатной программы, результирующий параметр с именем <имя>' склеивается с соответствующим входным параметром.

<обозначения результирующих параметров>::=
<имя результата>[,<обозначения результирующего параметров>]
<описания типов результирующих параметров> ::=
<изображения набора типов>
<изображения набора типов>::=
<изображение типа>[,<изображение набора типов>]

Отсутствие имен результирующих параметров для ветви результатов означает, что любое завершение по этой ветви в теле предиката реализуется в функциональной форме, т.е. вычислением списка выражений, число которых равно числу результирующих параметров по ветви. При наличии имен результирующих параметров может быть выбрана как предикатная, так и функциональная форма, причем такой выбор производится независимо по каждой ветви.

С гиперфункцией связана конструкция нового вида — предложение **расщепления**. Имеется **заголовок расщепления**, в большинстве случаев, это вызов гиперфункции. К каждой ветви вызова гиперфункции прикрепля-

ется предложение — **альтернатива расщепления**, исполняемая в случае завершения исполнения вызова этой ветвью гиперфункции.

Результатом исполнения тела предиката должна быть также указана ветвь гиперфункции, которой завершается вычисление предиката. Для обозначения того, что i-я ветвь — результат исполнения тела предиката, будем использовать конструкцию **завершителя ветви предиката** #i. Завершитель ветви должен быть указан в каждой позиции завершения исполнения тела предиката. Однако если гиперфункция является функцией (число ветвей равно 1), завершитель ветви может быть опущен.

Пример 2.3. Рассмотрим гиперфункцию elemTwo. Ее результатом является второй элемент e последовательности s целых чисел. Если s состоит менее, чем из двух элементов, реализуется вторая ветвь гиперфункции elemTwo, и тогда результатом второй ветви будет число элементов len.

В теле elemTwo будем использовать вызов стандартного предиката Comp(s: | e,r). Первая ветвь Comp реализуется для пустой последовательности s, а во второй ветви: e — начальный элемент, r — последовательность, полученная из s удалением начального элемента. Используется предложение расщепления следующего вида:

```
split Comp(s: | e1,s1) do 1: A | 2: B end
```

Предложение исполняется следующим образом. Вычисляется вызов предиката Comp. Если последовательность s — пуста, то реализуется первая ветвь Comp и затем выполняется предложение A, являющееся альтернативой предложения расщепления с меткой “1:”. В противном случае реализуется вторая ветвь Comp с вычислением e1 и s1, после чего исполняется предложение B, помеченное меткой “2:”. Дадим теперь определение предиката elemTwo:

```
elemTwo(seq int s: int e | nat len)
    int e1, e2; seq int s1, s2;
    split Comp(s: | e1,s1) do
    | 1:          len = 0 #2
    | 2:          split Comp(s1: | e2,s2) do
                  | 1:       len = 1 #2
                  | 2:       e = e2 #1
                  end
    end
fin
```

Завершитель ветви предиката #1 во второй альтернативе внутреннего оператора расщепления указывает, что эта альтернатива завершается первой ветвью предиката elemTwo.

3. ПРЕДИКАТНАЯ ПРОГРАММА

Предикатная программа содержит набор вычислимых **определений предикатов**, среди которых могут находиться описания типов и глобальных переменных:

<предикатная программа> ::=
{ <описание типа>;| <описание глобальных переменных>;|
<определение предиката> }
<описание глобальных переменных>::= **global** <описание переменных>

Переменные, описания которых встречаются перед определениями предикатов, называются **глобальными**. В нижеследующих определениях предикатов эти переменные являются входными параметрами, которые для сокращения записи не упоминаются при описании параметров и не указываются в вызовах предикатов.

Предикатная программа есть полная система вычислимых определений предикатов для предиката, представляющего спецификацию задачи. В полной системе определений для любого использующего вхождения имени непримитивного предиката существует его определение.

Система определений предикатной программы обычно является рекурсивной. При исполнении программы реализуется рекурсивное исполнение определений предикатов. Необходимым требованием является правильное построение рекурсии в системе определений, так чтобы процесс исполнения был бы конечным.

Исполнение предикатной программы — это исполнение не принадлежащего программе вызова предиката с аргументами, задаваемыми некоторым внешним по отношению к программе образом. Исполнение вызова предиката реализует исполнение определения этого предиката. Исполнение программы завершается при завершении исполнения определения предиката и присваивания итоговых значений результирующим переменным вызова предиката.

По умолчанию, предикатом, с которого начинается исполнение программы, является предикат с именем main.

Предикатная программа использует совокупность имен и обозначаемых ими объектов: предикатов, типов, переменных, полей кортежей и селекто-

ров объединений (см. разд. 6). Предикатная программа в целом определяет область глобальных имен: предикатов, глобальных переменных, типов, полей и селекторов. Глобальные имена могут обозначать не более одного объекта. Использование имени для двух разных объектов в глобальной области недопустимо.

Каждое описание предиката определяет автономную систему локализации имен переменных: параметров и локальных переменных. В описании предиката одно имя не может использоваться для двух разных переменных. Если глобальное имя используется в определении предиката, то это имя не может использоваться в качестве имени локальной переменной или параметра.

4. ОПЕРАТОРЫ

Одним из атрибутов вхождения конструкции Z в качестве подконструкции в конструкции G является наличие или отсутствие значения как результата исполнения Z. Будем говорить, что вхождение Z в G является **позицией значения**, если в соответствии с правилами исполнения G в этой позиции требуется значение. Если значение не требуется, будем говорить об **операторной позиции**.

Предложение в позиции значения в случае нормального завершения исполнения должно вырабатывать список значений (обычно, одно значение). Число значений и их типы должны соответствовать позиции. Для предложения в позиции значения должно существовать его нормальное завершение.

Предложение в операторной позиции не может вырабатывать значения. Вхождение оператора, отличного от предложения, разрешено в операторной позиции и недопустимо в позиции значения.

```
<предложение> ::=
        <предикат равенства> |
        <описание переменной с инициализацией> |
        <блок> |
        <условное предложение> |
        <предложение выбора> |
        <предложение расщепления>
<оператор> ::=
        <предложение> |
        <вызов предиката> |
        <параллельный оператор> |
```

<конструктор массива> |
<оператор разложения структурной переменной>

Конструктор массива определен в разд. 7, а операторы разложения – в разд. 6.

<предикат равенства> ::= <переменная> = <выражение>

Переменная в предикате равенства является результатом. При исполнении данного предложения переменной присваивается значение выражения. Результатом конструкции в позиции значения является присвоенное значение переменной.

```
<блок> ::=      [{] <тело блока> [}] |
                [{] [<тело блока>;] <список выражений> [}]
<список выражений>::=<выражение>[,<список выражений>]
<тело блока> ::=    <описание переменных> [;<тело блока>] |
                          <оператор> [;<тело блока>]
<описание переменных> ::=
            <описание переменной с инициализацией> |
            <изображение типа><пробел><имя>[{,<имя>}]
<описание переменной с инициализацией> ::=
            <изображение типа><пробел><имя> [= <выражение>]
```

Переменные, описанные в блоке, являются **локальными** в теле предиката, содержащего блок. Их описание действует в пределах тела предиката.

В цепочке операторов блока каждый следующий оператор использует значения локальных переменных, присвоенных предыдущими операторами. Каждая пара соседних операторов в цепочке должна быть связана хотя бы одной локальной переменной; если такой связи нет, их композиция должна быть оформлена параллельным оператором.

Описания и операторы блока исполняются в порядке их следования. Результатом исполнения описания переменных является создание в памяти одной или нескольких переменных, имена которых указаны в описании. При наличии инициализации вычисленное значение <выражения> присваивается переменной. Описание переменной с инициализацией “<изображение типа> z = <выражение>” используется как сокращение для конструкции вида “<изображение типа> z; z=<выражение>”. Значением конструкции описания переменной с инициализацией при использовании ее в позиции значения является присвоенное значение переменной.

Блок обрамляется фигурными скобками, которые могут опускаться, если блок используется в позициях следующих конструкций: условного

предложения, предложения выбора, предложения расщепления, параллельного оператора или конструктора массива.

Блок в позиции значения должен завершаться <списком выражений>, их число и типы должны соответствовать позиции.

<условное предложение>::=
if <выражение> **then** <сегмент>
[{ **elsif** <выражение> **then** <сегмент> }]
else <сегмент>
end

Условное предложение может содержать произвольное число фрагментов: **elsif** <выражение> **then** <сегмент>. Выражения после **if** и **elsif** должны быть логического типа.

Исполнение условного предложения начинается вычислением условия – выражения после **if.** При истинном значении исполняется <сегмент> после **then**. При ложном значении условия и при наличии **elsif** вычисляется следующее за ним условие и реализуется либо исполнение <сегмента> после **then**, либо следующего **elsif.** Если все условия оказались ложными, исполняется <сегмент> после **else**. Окончание исполнения выбранного сегмента после **then** или **else** определяет завершение исполнения условного предложения.

<сегмент> ::=
<блок> [<завершитель ветви предиката>] |
<оператор> <завершитель локального сегмента>
<завершитель ветви предиката> ::= #<метка ветви предиката>
<завершитель локального сегмента> ::=
##<метка альтернативы расщепления>
<метка ветви предиката> ::= <натуральное>
<метка альтернативы расщепления> ::= <натуральное>

Если исполнение <блока> в позиции <сегмента> нормально завершилось получением списка значений, то этот список является результатом предложения, в позиции которого находится <сегмент>, и типы списка значений должны соответствовать позиции <сегмента> в предложении. При отсутствии <завершителя ветви предиката> происходит нормальное завершение предложения, содержащего <сегмент>.

Завершитель ветви предиката определяет ненормальное завершение исполнения <сегмента> и всей иерархии конструкций, содержащих <сегмент>, в текущем исполняемом определении предиката. При этом исполнение определения предиката завершается ветвью с указанным номером. Ес-

ли исполнение <сегмента> завершилось списком значений, то эти значения присваиваются результирующим параметрам в определении предиката по этой ветви, причем типы списка значений должны соответствовать типам параметров.

Сегмент вида <оператор> <завершитель локального сегмента> допустим в заголовке предложения расщепления. После исполнения <оператора> реализуется переход на начало альтернативы расщепления с указанным номером для наименьшего объемлющего предложения расщепления, содержащего альтернативу с указанным номером.

<предложение выбора>::=
case <выражение> **of** [⊥]
<альтернативы выбора>
[**else** <сегмент>]
end

Предложение выбора является другой формой условного предложения. Выражение после **case** называется **селектором** и должно быть **интегрального** типа (совместимого с типом **int**).

<альтернативы выбора>::=
<метка альтернативы выбора> **:** <сегмент>
{⊥ <альтернативы выбора>}
<метка альтернативы выбора>::=
<список частей подмножества типа>

Метка альтернативы определяет либо одно константное значение, либо константный диапазон, либо список константных значений и диапазонов (см. разд. 6). Константы в метках должны соответствовать типу селектора. Подмножества, определяемые двумя разными метками, не могут пересекаться.

Исполнение предложения выбора реализуется следующим образом. Вычисляется селектор, и его значение последовательно сравнивается со значениями меток альтернатив; если метка представлена подмножеством, проверяется попадание значения в это подмножество. Исполняется альтернатива, метка которой соответствует значению селектора. Если ни одна метка не соответствует значению селектора, исполняется сегмент после **else**. Если набор значений меток полностью покрывает тип <выражения>, альтернатива **else** может отсутствовать**.**

<параллельный оператор>::=
<оператор>||<продолжение параллельного оператора>
<продолжение параллельного оператора>::=
<оператор> |
<параллельный оператор>

Параллельный оператор состоит из произвольного числа операторов, разделенных "||", которые выполняются независимо друг от друга. Выполнение параллельного оператора заканчивается при завершении исполнения всех его составляющих. Исполнение завершителя ветви предиката в одном из операторов, входящих в параллельный оператор, инициирует прекращение исполнения других составляющих параллельного оператора. Операторы, образующие параллельный оператор, должны иметь непересекающиеся наборы результирующих переменных. Предложения в составе параллельного оператора не могут вырабатывать значения.

<вызов предиката> ::=
<имя предиката>([<аргументы>]:<результаты>) |
<выражение>([<аргументы>]:<результаты>)

Значением <выражения> в позиции вызова предиката должен быть некоторый предикат.

<аргументы> ::= <список выражений>
<результаты> ::=
[<результаты ветви>] [<завершитель>] [⊥ <результаты>]
<завершитель>::= <завершитель ветви предиката> |
<завершитель локального сегмента>

<завершитель> определен выше при описании конструкции <сегмент>. Завершитель ветви предиката указывает на завершение исполнения определения предиката, содержащего данный вызов. Завершитель локального сегмента определяет завершение заголовка расщепления, частью которого является вызов предиката, и переход на исполнение альтернативы расщепления, номер которой указан в завершителе.

<результаты ветви>::=
[<изображение типа><пробел>] <переменная> [,<результаты ветви>]

Вхождение <изображения типа> перед результирующей переменной означает ее описание в качестве локальной переменной тела предиката, в котором находится данный вызов предиката. Описываемая переменная создается в памяти перед исполнением вызова предиката. Описание результи-

рующей переменной внутри вызова эквивалентно описанию этой переменной перед вызовом.

Исполнение вызова предиката реализуется следующим образом. Если в позиции имени предиката находится выражение, то вызываемый предикат определяется как значение этого выражения. Результатом исполнения аргументов вызова является набор значений. Вычисление каждого аргумента вызова реализуется независимо от вычисления других, т.е. параллельно. Полученный набор значений аргументов присваивается соответствующим входным параметрам определения вызываемого предиката. Далее исполняется тело определения вызываемого предиката; в случае примитивного предиката реализуются действия, предусмотренные его семантикой. В процессе исполнения тела определяется итоговая ветвь предиката, и вычисляются значения результирующих переменных по этой ветви. Наконец, полученные значения результирующих переменных присваиваются соответствующим результирующим переменным вызова предиката, и исполнение вызова завершается по этой ветви вызова. Отметим, что подстановка аргументов и результатов предиката реализуется по значению.

<предложение расщепления>::=
split <заголовок расщепления> **do** [⊥]
<альтернативы расщепления> **end**
<заголовок расщепления>::=<оператор>
<альтернативы расщепления>::=
<метка альтернативы расщепления>**:** <сегмент>
[⊥ <альтернативы расщепления>]
<метка альтернативы расщепления> ::= <натуральное>

Исполнение расщепления начинается с заголовка. Для заголовка недопустимо нормальное завершение — исполнение заголовка всегда заканчивается некоторым завершителем. Если это завершитель локального сегмента вида ##i, то реализуется переход на альтернативу расщепления, помеченную меткой i (альтернатива с такой меткой должна существовать). Исполнение предложения расщепления завершается по окончании исполнения выбранной альтернативы расщепления. Предложение в качестве заголовка расщепления не может вырабатывать значения.

Если заголовком предложения расщепления является вызов предиката и для i-й ветви результатов вызова определен завершитель локального сегмента ##i, то этот завершитель может быть опущен. Аналогичное умолчание действительно для операторов разложения объединения, последовательности или множества (см. разд. 6), используемых в качестве заголовка расщепления.

Предложение расщепления с единственной альтернативой расщепления вырождается в блок или параллельный оператор. Поэтому будем считать, что число альтернатив в расщеплении не менее двух.

Пример 4.1. Рассмотрим более простую версию предиката elemTwo из примера 2.3. Гиперфункция elemTwo на первой ветви получает второй элемент e последовательности s целых чисел. На второй ветви нет результатов. Она реализуется в случае, если s состоит менее, чем из двух элементов. Упрощая определение elemTwo из примера 2.3, получим:

```
elemTwo(seq int s: int e | )
        split Comp(s: ##1 | int  e1, seq int s1 ##2) do
        | 1:          #2
        | 2:          split Comp(s1: ##1 | e, seq int s2 ##2) do
                      | 1:        #2
                      | 2:        #1
                      end
        end
fin
```

Поскольку завершители локальных сегментов ##1 и ##2 в двух вызовах Comp находятся соответственно на первой и второй ветвях вызова, они могут быть опущены. Далее, альтернативы внутреннего предложения расщепления и первая альтернатива внешнего фактически пусты и используются лишь для размещения завершителей ветвей предиката. В этом случае, пустые альтернативы удаляются, а завершители помещаются в соответствующие позиции ветвей вызова Comp:

```
elemTwo(seq int s: int e | )
        split Comp(s: #2 | int  e1, seq int s1) do
        | 2:          Comp(s1: #2 | e, seq int s2 #1)
        end
fin
```

В приведенном определении предложение расщепления вырождается, поскольку содержит только одну альтернативу. Два вызова Comp связаны переменной s1, которая получается в первом и используется во втором вызове. Поэтому предложение расщепления заменяется блоком:

```
elemTwo(seq int s: int e | )
        Comp(s: #2 | int  e1, seq int s1);
        Comp(s1: #2 | e, seq int s2 #1)
fin
```

Если бы между двумя вызовами не было связи, результатом замены был бы параллельный оператор.

5. ВЫРАЖЕНИЯ

Результатом исполнения выражения является значение или набор значений. Набор более чем из одного значения допустим лишь в соответствующей позиции аргумента предиката или вызова функции либо как результат тела предиката в функциональном стиле.

<выражение>::= <первичное> |
<выражение><знак бинарной операции><выражение> |
<знак унарной операции><выражение>

При записи выражений используются правила приоритетов операций, что позволяет опускать круглые скобки.

Набор операций в данной версии языка Р в основном заимствован из языка С++.

<знак унарной операции>::= + | - | !

Операция “!” есть логическое отрицание.

<знак бинарной операции>::=
* | / | % | + | - | << | >> |
< | > | <= | >= | = |!= | & | ^ | ⊥ | && | **or** | **in** | \

Операции “*” и “/” определяют умножение и деление интегральных, вещественных и комплексных операндов. Операция % определяет остаток от целочисленного деления. Если операнд $b \neq 0$, то (a/b)*b+a%b = a. Если оба операнда операции “%” являются неотрицательными, то результат неотрицательный.

Для операндов операций “+” и “-” возможно предварительное приведение значений операнда к наибольшему типу. Операция “+” используется также для объединения множеств типа **set**, а также для добавления элемента к множеству. Кроме того, операция “+” используется для объединения

массивов с одинаковым типом элементов и непересекающимися типами индексов.

Операции "<<" и ">>" определяют соответственно сдвиг интегрального значения налево и направо. Второй операнд должен быть неотрицательным. Сдвиг налево определяет умножение первого операнда на 2 в степени второго операнда, сдвиг направо — целочисленное деление первого операнда на 2 в степени второго операнда.

Операции "<", ">", "<=", ">=", "=" и "!=" определяют отношения для арифметических операндов; результат — логическое значение. Операция "!=" обозначает отношение неравенства. Операции "&", "|" и "^" определяют соответственно побитовые операции "и", "или" и "дополнительное или". Операция "&&" определяет логическое "и"; операция **or** — логическое "или".

Операция e **in** X определяет принадлежность элемента e множеству X типа **set**. Операция "\" используется для вычитания второго множества из первого, а также для удаления элемента из множества.

Приоритет операций следующий:

- унарные операции +, — и !
- *, / и %
- бинарные + и –
- << и >>
- **in** и \
- <, >, <= и >=
- = и !=
- &
- ^
- |
- &&
- **or**

```
<первичное>::=    <переменная> |
                  <изображение константы> |
                  <вызов функции> |
                  <предложение> |
                  <имя предиката> |
                  <порождение предиката> |
                  <производные формы конструктора массива> |
                  <конструктор структурного объекта> |
                  (<выражение>)
```

Описание производных форм конструктора массива см. в разд. 7, конструктора структурного объекта — в разд. 6.

Значением конструкции <имя предиката> является обозначаемый именем предикат. Это значение может быть присвоено переменной типа "предикат" через <предикат равенства>.

Исполнение <предложения> завершается вычислением <списка выражений>, находящегося в составе <предложения>. Число выражений и их типы должны соответствовать позиции вхождения <предложения>.

<порождение предиката> ::=
    lambda (<описание исходных и результирующих параметров>)
    [{] <тело предиката> [}]

Исполнение конструкции порождения предиката реализует построение определения нового предиката, правой частью которого будет <оператор> тела предиката при фиксации в нем значений свободных переменных на момент начала исполнения конструкции порождения предиката. Новый предикат становится результирующим значением конструкции порождения предиката.

<переменная> ::= <простая переменная> |
    <элемент массива> |
    <поле переменной>

<простая переменная> ::= <имя переменной>
<имя переменной> ::= <имя> | <имя склеиваемого результата>
<элемент массива> ::= <переменная>[<выражение>]
<поле переменной> ::= <переменная>.<имя поля>
<имя поля> ::= <имя>

Вхождение переменной в качестве результирующей переменной вызова предиката (в том числе, предиката равенства и других стандартных предикатов, имеющих специальные обозначения) называется **определяющим**. Остальные вхождения переменной в теле предиката называются **использующими**.

Результатом исполнения использующего вхождения переменной является значение переменной. Определяющие вхождения допустимы для переменной с индексами в теле конструктора массива и для простой переменной.

<вызов функции> ::= <имя предиката>(<аргументы>) |
    <выражение>(<аргументы>)

Значением <выражения> в позиции вызова предиката должен быть некоторый предикат. Вызываемый предикат должен иметь одну ветвь, т.е. быть функцией.

Исполнение вызова функции аналогично исполнению вызова предиката. По завершению исполнения определения предиката формируется значение вызова функции: это значение результирующей переменной определения предиката или набор значений, если результирующих переменных несколько.

6. ТИПЫ

<описание типа> ::=
type <имя типа>[(<описания или обозначения параметров>)] =
<изображение типа> |
<предописание типа>
<имя типа> ::= <имя>

Описание типа связывает имя типа с его изображением. Правила для <описания или обозначения параметров> даны в разд. 2.

<предописание типа> ::= **type** <имя типа>

Если имя типа используется до его описания, что возможно для рекурсивных определений типов, то перед первым вхождением имени оно должно быть декларировано с помощью предописания.

Тип может быть **параметризован**, т.е. зависеть от набора значений переменных, указанных в качестве параметров.

<изображение типа> ::= <изображение примитивного типа> |
<изображение подмножества типа> |
<имя типа>[(<аргументы>)] |
<изображение типа предиката> |
<изображение произвольного типа> |
<изображение структурного типа>
<изображение примитивного типа> ::=
nat | **int** | **real** | **bool** | **char** | **complex**

Описатель **nat** обозначает множество натуральных чисел, **char** — множество символов некоторого алфавита.

<изображение подмножества типа> ::=
{<описание переменных>:<выражение>} |
<изображение диапазона> |

{<список частей подмножества типа>} |
{<изображение подмножества типа кортеж>}

Конструкция {T x: <выражение>} определяет подмножество типа T. Описание переменной x действует в пределах этой конструкции. Множество значений переменной x, для которых логическое <выражение> имеет значение “истина”, является определяемым подмножеством типа T. Если <выражение> содержит другие переменные, то все они являются **параметрами** изображаемого типа.

Если <описание переменных> определяет более одной переменной, то конструкция {<описание переменных>:<выражение>} определяет подмножество типа “кортеж”. В качестве полей кортежа используются локальные переменные, перечисляемые в <описании переменных>.

В качестве примера рассмотрим изображение типа для диапазона натуральных чисел от 1 до n + 1, где n является параметром:

{**nat** i: i>=1 && i<=n+1}

Для диапазонов будем использовать специальную синтаксическую форму:

<изображение диапазона> ::= <выражение>..<выражение>

Изображение типа для приведенного примера может быть дано в более компактном виде: 1..n+1.

<список частей подмножества типа> ::=
<выражение> [,<список частей подмножества типа>] |
<изображение диапазона> [,<список частей подмножества типа>]

Вычисленное значение <выражения> определяет одно значение, принадлежащее подмножеству типа. Определяемое подмножество объединяет значения всех частей списка.

<изображение подмножества типа кортеж> ::=
<изображение подмножества типа поля>
{;<изображение подмножества типа кортеж>}
<изображение подмножества типа поля> ::=
<список частей подмножества типа> |
<изображение диапазона>

Данная конструкция определяет подмножество типа “кортеж” как произведение типов, представленных в виде изображения подмножества для каждого из полей кортежа.

Если изображение подмножества типа используется в правой части описания типа, то параметры подмножества типа должны быть указаны как параметры описываемого типа. Например,

type Diapason(**int** n) = 1..n+1;

Для типа с параметрами допускается вводить обозначение без параметров в случае, когда в качестве параметров используются глобальные переменные (см. разд. 3). Например, если n – глобальная переменная, возможно следующее обозначение:

type DIAP = Diapason(n);

Параметризованный тип может быть использован для определения другого типа непосредственно в виде <имя типа>(<аргументы>) или в составе структурного типа. Если в <аргументах> используются переменные, то эти переменные должны быть параметрами определяемого типа.

<изображение типа предиката> ::=
predicate ([<изображение типов аргументов>]:
<изображение типов результатов>)
<изображение типов аргументов> ::= <изображение набора типов>
<изображение типов результатов> ::=
[<изображение набора типов>] [⊥ <изображение типов результатов>]

Изображение типа предиката используется для описания переменной предикатного типа.

<изображение произвольного типа> :: **type**

Описатель **type** специфицирует следующие за ним параметры предиката как переменные некоторого произвольного типа.

Структурный тип определяется в виде композиции других типов, называемых **компонентными** по отношению к структурному. Структурными типами являются: массив, кортеж, объединение, последовательность и множество подмножеств.

<изображение структурного типа> ::=
<изображение типа массива> |
<изображение типа кортежа> |
<изображение типа объединения> |
<изображение типа последовательности> |
<изображение типа множества>

<изображение типа массива> ::= array <тип индексов> of <тип элементов>
<тип индексов> ::= <изображение типа>
<тип элементов> ::= <изображение типа>

Значение массива состоит из совокупности элементов. Каждый элемент доступен по индексу в массиве. Тип индексов должен быть конечным. Детальное описание представлено в разд. 7.

<изображение типа кортежа> ::=
struct <изображение типов полей> **end**
<изображение типов полей> ::=
<изображение типа поля>[<пробел> <список имен полей>]
[,<изображение типов полей>]
<изображение типа поля> ::= <изображение типа>
<список имен полей> ::= <имя поля>[,<список имен полей>]

Тип кортежа является произведением типов компонент. Значение типа кортежа состоит из совокупности значений компонентных типов.

Примитивный тип **complex** рассматривается как тип, определяемый изображением **struct real** re, im **end.**

<изображение типа объединения> ::=
union <изображение типов альтернатив> **end**
<изображение типов альтернатив> ::=
[<имя селектора>:] [<изображение типов полей>]
[|<изображение типов альтернатив>]
<имя селектора> ::= <имя>

Тип объединения представлен в виде объединения альтернатив, в общем случае, кортежей. Значение типа объединения есть значение одной из альтернатив. Альтернатива является **пустой** при пустом списке полей альтернативы.

<изображение типа последовательности> ::=
seq <изображение типа элемента>
<изображение типа элемента> ::= <изображение типа>

Последовательность определяется как упорядоченная совокупность элементов, возможно пустая. Тип **seq char** может изображаться описателем **string**.

<изображение типа множества> ::= **set** <изображение базового типа>
<изображение базового типа> ::= <изображение типа>

Тип множества есть множество всех подмножеств некоторого конечного базового типа. Для множеств определены операции сложения множеств “+”, вычитания множеств “\” и удаления элемента из множества “\” (см. разд. 5).

Структурный тип определяется двумя **фундаментальными предикатами: конструктором**, определяющим построение значения структурного типа по набору значений компонентных типов, и **разложением**, определяющим набор соответствующих компонентов для объекта структурного типа.

```
<конструктор структурного объекта> ::=
                <конструктор кортежа> |
                <конструктор объединения> |
                <конструктор последовательности> |
                <конструктор множества>
<оператор разложения структурной объекта> ::=
                <разложение кортежа> |
                <разложение объединения> |
                <разложение последовательности> |
                <разложение множества>

<конструктор кортежа> ::=
                <тип кортежа>(<выражения полей кортежа>)
<тип кортежа> ::= <изображение типа>
<выражения полей кортежа> ::=
                [<имя поля>:]<выражение>[,<выражения полей кортежа>]
```

Конструктор кортежа вычисляет значение типа кортеж, объединяя значения, полученные при вычислении выражений для всех полей кортежа. Выражения для полей кортежа могут вычисляться параллельно.

Естественно соглашение, что для элемента массива вида A[<тип>(x,y)] используется традиционное написание A[x,y]. Данное правило применяется и для большего числа измерений массива.

```
<разложение кортежа> ::=
            <объект-кортеж> -> (<компонентные переменные кортежа>)
<объект-кортеж> ::= <выражение>
<компонентные переменные кортежа> ::=
            [<изображение типа><пробел>]<имя переменной>
            [,<компонентные переменные кортежа>]
```

Оператор разложения кортежа обеспечивает доступ к компонентным переменным, ассоциированным со значением <объект-кортежа>. Компонентные переменные являются результирующими, т.е. их вхождения являются определяющими. При наличии <изображения типа> <пробел> вхождение компонентной переменной является также ее описанием.

Для объектов типа **complex** может быть использован конструктор кортежа **complex**(<выражение>, <выражение>) и соответствующий оператор разложения.

<конструктор объединения> ::=
<тип объединения>(<имя селектора>: [<выражения полей кортежа>])
<тип объединения> ::= <изображение типа>

Конструктор объединения формирует значение типа объединения как альтернативу, идентифицируемую именем селектора, со значениями полей для непустой альтернативы.

<разложение объединения> ::=
<объект-объединение> -> (<переменные альтернатив объединения>)
<объект-объединение> ::= <выражение>
<переменные альтернатив объединения> ::=
[<имя селектора>:]
[<компонентные переменные кортежа>][<завершитель>]
[|<переменные альтернатив объединения>]

Оператор разложения обеспечивает доступ к переменным полей альтернатив, ассоциированным со значением <объект-объединения>. Ограничитель "|" разделяет альтернативы объединения. Оператор разложения объединения аналогичен вызову предиката-гиперфункции. Результирующие переменные ветвей гиперфункции являются переменными альтернатив объединения. Для каждой альтернативы объединения определяются переменные, являющиеся полями кортежа этой альтернативы. Для пустой альтернативы возможно присутствие лишь завершителя. Вхождения компонентных переменных в операторе разложения объединения являются их определяющими вхождениями и описаниями (если <изображение типа> присутствует).

В общем случае разложение объединения используется в качестве заголовка предложения расщепления. Завершитель локального сегмента может быть опущен, если в других альтернативах используются завершители ветвей предиката (в этом случае расщепление вырождается), либо если оператор разложения объединения является заголовком предложения расщепле-

ния и номер локального завершителя совпадает с номером ветви гиперфункции.

```
<конструктор последовательности>::=
    <тип последовательности>() |
    <тип последовательности>
    (<аргументы конструктора последовательности>)|
    <изображение строковой константы>
<тип последовательности>::=<изображение типа>
```

Конструктор определяет значение типа последовательности. В первом случае конструктор определяет пустую последовательность. Остальные случаи определены ниже. Изображение строковой константы в качестве конструктора допускается только для типа **string**.

```
<аргументы конструктора последовательности> ::=
            <список элементов> [, <последовательность-продолжение>] |
            <последовательность-начало>, <список элементов>
<список элементов> ::= <список выражений>
<последовательность-продолжение> ::= <выражение>
<последовательность-начало> ::= <выражение>
```

Последовательность конструируется из элементов и других последовательностей. Тип выражения, используемого в качестве аргумента последовательности, должен соответствовать либо типу элемента, либо типу конструируемой последовательности.

```
<разложение последовательности> ::=
        <объект-последовательность> ->
        ( [<завершитель ветви>] ⊥ <разложение по второй ветви>
                                        [<завершитель ветви>] )
<разложение по второй ветви> ::=
       [<изображение типа><пробел>]<переменная элемента>,
       [<изображение типа><пробел>]<переменная последовательности> |
       [<изображение типа><пробел>]<переменная последовательности>,
       [<изображение типа><пробел>]<переменная элемента>
<объект-последовательность>::=<выражение>
<переменная последовательности>::=<имя переменной>
<переменная элемента>::=<имя переменной>
```

Оператор разложения последовательности декларирует, что либо последовательность пуста, и тогда реализуется первая ветвь гиперфункции, либо разлагаемая последовательность состоит из элемента и подпоследовательности, причем возможны два способа разложения: элемент + последо-

вательность и последовательность + элемент. В остальном, разложение последовательности аналогично разложению объединения.

```
<конструктор множества> ::=
    <тип множества>([<список элементов множества>])
<список элементов множества> ::= <список выражений>
```

Конструктор множества определяет множество через список его элементов. Пустой список элементов обозначает пустое множество. Тип выражений для элементов должен быть совместимым с базовым типом множества.

```
<разложение множества> ::=
        <объект-множество> ->
        (  [<завершитель ветви>] ⊥
         [<изображение типа><пробел>]<переменная элемента>,
         [<изображение типа><пробел>]<переменная остатка множества>
         [<завершитель ветви>] )
<объект-множество> ::= <выражение>
<переменная остатка множества> ::= <переменная>
```

Оператор разложения множества декларирует, что либо множество пусто, и тогда реализуется первая ветвь гиперфункции, либо множество представляется из элемента и остальной части множества, равной разлагаемому множеству за вычетом элемента. Способ, которым выделяется элемент из множества, не фиксируется. В остальном, разложение множества аналогично разложению объединения.

Набор описаний типов может быть рекурсивным. Рекурсивное вхождение определяемого типа возможно лишь в альтернативе типа объединения.

Пример 6.1. Тип Tree2 определяет регулярное двоичное дерево, каждая вершина которого содержит две дуги на поддеревья и атрибут типа T:

```
type  Tree2 =  union   s1: |
                       s2: struct T attr, Tree2 right, Tree2 left end
               end
```

7. МАССИВЫ

Массив состоит из совокупности элементов. Каждый элемент доступен по индексу в массиве. Тип индексов должен быть конечным. Элемент мас-

сива определяется конструкцией: <переменная>[<выражение>]. Значением конструкции <переменная> является массив, <выражение> определяет значение индекса.

Массив создается с помощью оператора “конструктор массива”.

```
<конструктор массива> ::=
                    forAll <итератор> do <тело конструктора массива> end |
                    <конструктор массива по частям типа индексов>
<итератор> ::=      <простая переменная> in <изображение типа> |
                    <простая переменная>=<выражение>..<выражение>
<тело конструктора массива> ::=
                    <оператор> |
                    <элемент массива> = <выражение>
```

Исполнение конструктора массива реализуется следующим образом. Тип, используемый в итераторе, обозначим через I; во втором случае — это тип диапазона. В памяти создается конструируемый массив как значение типа **array** I **of** T, где T — тип элементов. Для переменной, указанной в <итераторе>, реализуется итерация по множеству I. Для каждого значения i итерируемой переменной выполняется тело конструктора между **do** и **end**, реализующего вычисление элемента. Вычисление тела конструктора для разных индексов i реализуется независимо, т.е. параллельно. Конструктор завершает свою работу при завершении вычисления тела конструктора для всех индексов. Предложение в качестве тела конструктора не может вырабатывать значения.

Конструктор массива по частям типа индексов определен ниже.

Оператор **forAll** определяет общую форму конструктора массива. Определим частные случаи конструктора массива (его производные формы), для которых используются специальные обозначения.

```
<производные формы конструктора массива> ::=
                        <поэлементное определение массива> |
                        <объединение двух массивов> |
                        <замещение элемента> |
                        <вырезка массива>
```

Перечисленные конструкции являются выражениями, порождающими массив в качестве результирующего значения.

```
<поэлементное определение массива> ::=
                        [<тип массива>](<список выражений>)
<тип массива> ::= <изображение типа>
```

Поэлементное определение массива используется для строго упорядоченного непараметрического типа индексов. Каждое выражение в списке определяет один элемент массива. Выражения для элементов перечисляются в порядке следования индексов. Длина списка должна совпадать с числом элементов массива. Выражения в списке выражений могут вычисляться параллельно. <тип массива> может быть опущен в случаях, когда этот тип однозначно определяется по позиции, в которой находится данный конструктор.

<объединение двух массивов> ::=
<выражение-массив>+<выражение-массив>
<выражение-массив> ::= <выражение>

Оба операнда операции объединения "+" должны вычислять значения типа "массив". Типы объединяемых массивов должны иметь совпадающие типы элементов и непересекающиеся типы индексов, причем типы индексов должны принадлежать некоторому общему типу. Тип индексов результирующего массива есть объединение типов индексов операндов, а тип элементов тот же, что и у операндов. Элементы результирующего массива есть объединение элементов массивов-операндов.

<замещение элемента> ::=
<выражение-массив>[<выражение-индекс> ! <выражение-элемент>]
<выражение-индекс> ::= <выражение>
<выражение-элемент> ::= <выражение>

Сначала параллельно вычисляются три выражения операции замещения элемента. Операция создает новый массив, являющийся копией массива, полученного вычислением <выражения-массив>. В копии массива заменяется один элемент, новое значение которого определяется <выражением-элемент> с индексом <выражение-индекс>.

<вырезка массива> ::= <выражение-массив>[<суженный тип индексов>]
<суженный тип индексов> ::= <изображение типа>

Результатом вычисления <выражения-массива> является массив. Его тип индексов должен содержать <суженный тип индексов>. Создается новый массив, значением которого является часть исходного вычисленного массива, спроецированная на <суженный тип индексов>.

Конструктор массива по частям типа индексов является гибридом конструктора массива и объединения массивов.

<конструктор массива по частям типа индексов**>** ::=
forCase <итератор> **do** <тело конструктора массива>
{ **elseCase** <итератор> **do** <тело конструктора массива> }
end

Допустим, итераторы в **forCase** и во всех **elseCase** определены для типов индексов I_1, I_2, ... , I_n. Эти типы должны не пересекаться между собой и быть подмножествами некоторого одного типа. Допустим, тип I есть объединение перечисленных типов. Конструктор определяет некоторый массив R, его тип индексов есть I. В каждом из операторов после ограничителей **do** массив R определяется для соответствующего подтипа индексов I. Исполнение частей конструктора после **forCase** и всех **elseCase** реализуется параллельно.

Поскольку сумма элементов массива является часто встречающейся операцией, введем для суммы элементов массива X специальное обозначение: SUM(i = A..B, X[i]: s), где A и B — произвольные выражения, а s — значение суммы.

8. ИМПЕРАТИВНОЕ РАСШИРЕНИЕ ЯЗЫКА

Конструкции языка, принадлежащие императивному расширению, недопустимы в предикатной программе, за исключением рекуррентного оператора. Эти конструкции возникают в программе в результате трансформаций предикатной программы.

Рекуррентный оператор может использоваться непосредственно в предикатной программе.

<рекуррентный оператор> ::=
<оператор>**;** <рекуррентный цикл for>

<оператор> инициализирует один или несколько начальных элементов массива, определяемого <рекуррентным циклом for>.

<рекуррентный цикл for> ::= **for** <заголовок> **do** <оператор> **end**
<заголовок> ::=
<простая переменная>=<выражение>..<выражение>[**step**<выражение>]

Если “**step**<выражение>” отсутствует, шаг перевычисления <переменной> равен 1.

В составе тела цикла for используются рекуррентные определения вида <элемент массива> = <выражение>.

Пример 8.1. Предикат Fib(**nat** n**:** Ar(n) f) определяет n чисел Фиббоначи в соответствии с формулой: $f_1 = 1$ & $f_2 = 1$ & $(\forall k>2)(f_k = f_{k-1}+f_{k-2})$.

```
type Ar(n) = array 1..n of nat;
Fib(nat n: Ar(n) f) ≡
            f[1] = 1; f[2] = 1;
            for i = 3..n do f[i] = f[i-1]+f[i-2] end
```

Остальные конструкции возникают в программе в результате трансформаций предикатной программы.

```
<общий оператор> ::=
        <оператор> |
        <оператор императивного расширения>
```

Вместо оператора (в произвольной позиции оператора в предикатной программе) в полученной императивной программе будет находиться общий оператор.

```
<оператор императивного расширения> ::=
        <групповой оператор присваивания> |
        <оператор присваивания> |
        <оператор while> |
        <оператор loop> |
        <оператор exit> |
        <оператор for> |
<групповой оператор присваивания> ::=
        ⌊<список переменных>⌋ := ⌊<список выражений>⌋
<список переменных> ::= <переменная>[,<список переменных>]
```

Исполнение группового оператора начинается определением (вычислением) списка переменных в левой части. Независимо (параллельно) вычисляется каждое из выражений в списке левой части. Значения из полученного набора одновременно присваивается соответствующим переменным в правой части.

```
<оператор присваивания> ::= <переменная> := <выражение>
```

Оператор присваивания может возникнуть в результате раскрытия группового оператора присваивания. Вычисление возможных индексных выражений в <переменной> не должно создавать побочных эффектов, влияющих на результат вычисления <выражения>.

```
<оператор loop> ::= loop <общий оператор> end
```

Выход из цикла реализуется по оператору **exit**, встречающегося в качестве одного из операторов в теле оператора **loop**.

<оператор exit> ::= **exit**
<оператор while> ::= **while** <выражение> **do** <общий оператор> **end**

Тип <выражения> в операторе while – логический.

<оператор for> ::= **for** <заголовок> **do** <общий оператор> **end**

В предикатной программе может использоваться оператор printf языка C.

9. ДОПОЛНИТЕЛЬНАЯ СЕМАНТИКА

Данный раздел не для пользователей. Если определение предиката соответствует правильно построенному логическому утверждению, то описываемые ниже правила должны быть выполнены.

Для произвольной конструкции G в теле предиката определим понятие **среза** тела предиката для конструкции G. Это часть тела предиката, доступная в процессе исполнения после исполнения конструкции G. Срезом для некоторой ветви вызова предиката-гиперфункции является часть тела предиката, доступная при завершении вызова этой ветвью. В общем случае срез имеет форму ациклического графа, т.е. графа без циклов.

При определении среза для описания локальной переменной z это описание перемещается в точку, ближайшую к определениям переменной z. Если определение единственно, то местом описания считается соответствующий предикат равенства или ветвь вызова предиката, где z является результирующей переменной. При наличии нескольких определений z местом описания считается ближайшая **точка ветвления**, из которой доступны все определения. Точка ветвления есть место в программе после исполнения следующих элементов:

– условия в условном предложении,

– выражения-селектора в предложении выбора,

– вызова предиката-гиперфункции с числом ветвей больше 1.

Отметим, что срез для определения переменной не может содержать другого определения этой переменной.

Область локализации локальной переменной находится внутри среза ее описания и ограничена совокупностью всех определяющих и использующих вхождений переменных. Область локализации переменной может быть шире блока, в котором находится описание переменной.

Переменная является **результатом** некоторой конструкции, если она определяется (присваивается) внутри и используется вне конструкции, т.е. либо она является результирующим параметром определения предиката, либо срез для данной конструкции содержит ее использующие вхождения. **Интерфейсом** конструкции является набор результирующих переменных при завершении исполнения конструкции. Интерфейс содержит набор собственных результатов конструкции, а также набор результатов всех предыдущих выполненных конструкций, которые используются в срезе для данной конструкции либо являются результирующими параметрами.

При любом завершении исполнения тела предиката некоторой ветвью предиката итоговый интерфейс должен совпадать со списком результирующих параметров этой ветви.

В точке **слияния** нескольких сегментов их интерфейсы должны совпадать. Слияние реализуется при одинаковом завершении нескольких альтернатив для следующих конструкций: условного предложения, предложения выбора или предложения расщепления. Завершения альтернатив являются одинаковыми, если все эти завершения являются либо нормальными, либо все их завершители совпадают, т.е. определяют одну и ту же ветвь предиката или одну и ту же альтернативу расщепления.

Операнды бинарной операции (и операции большей арности) не могут вычисляться параллельно в случае, когда результат одного из операндов используется в другом. Аналогичное ограничение на параллельность исполнения имеет место для аргументов вызова предиката или функции, если результат одного из аргументов используется в одном из следующих аргументов.

ЗАКЛЮЧЕНИЕ

По сравнению с работой [1] язык Р претерпел значительные изменения в результате его применения как языка программирования для небольшого числа задач. Изменения следующие:

- операторы теперь могут вырабатывать значение как результат оператора; вследствие этого, программу можно писать в стиле функционального программирования; кроме того, можно произвольно и гибко сочетать функциональный и операторные стили;
- появился новый структурный тип **set** <базовый тип>, определяющий множество подмножеств базового типа;

– унифицированы конструкторы и операторы разложения для структурных типов;
– появилась новая производная форма **forCase** для конструктора массива по частям типа индексов;
– отменен блочный принцип локализации переменных; областью локализации является все определение предиката;
– добавление апострофа к имени входного параметра обозначает результирующий параметр, «склеиваемый» с входным параметром без апострофа и др.

Очевидно, что общая структура предикатной программы в виде набора определений предикатов, описаний типов и глобальных переменных имеет временный характер. В дальнейшем неизбежно появятся средства модульной и объектно-ориентированной организации программы. Менее очевидны расширения в сторону задач системного программирования, спецификация которых принципиально не может быть представлена в виде математического предиката. Здесь необходимы специальные исследования.

В дальнейшем, в императивное расширение языка будут включены дополнительные средства для организации и синхронизации параллельных процессов, необходимых для случая, когда в параллельном операторе встречаются рекурсивные вызовы предикатов.

Достоинством подхода предикатного программирования является то, что правильность предикатной программы может обеспечиваться средствами ее автоматизированной верификации с применением всего математического аппарата. Реализация автоматизированной верификации имеет гораздо больше шансов на успех для предикатных программ, чем для императивных. В качестве первого шага в этом направлении планируется расширить язык Р средствами спецификации предикатов, определяемых в предикатной программе.

СПИСОК ЛИТЕРАТУРЫ

1. **Шелехов В.И.** Введение в предикатное программирование. — Новосибирск, 2002. — 82с. — (Препр. / ИСИ СО РАН; № 100).
2. **Wijngaarden A., Mailboux B.J., Peck J.E.L., Koster C.H.A.** Bericht uber die algorithmische sprache ALGOL 68. — Berlin: Akademie Verlag, 1972. — 381p.
3. **Мир** Лиспа / Том 2. Методы и системы программирования / Пер. с англ. – М.: Мир, 1990. – 318с.
4. **Gilmore S.** Programming in Standard ML ’97: A Tutorial Introduction. — 1997. — 68p. — (Prepr./University of Edinburg; ESC-LFCS-97-364)

5. **Бирюкова Ю.В.** Sisal 90. Руководство пользователя. — Новосибирск, 2000. — 83с. — (Препр. / ИСИ СО РАН; N 72.)
6. **Armstrong J.L., Virding S.R., Williams M.C.** Erlang. User's Guide & Reference Manual. Version 3.2. – Ellemtel Utvecklings AB, 1991. – 45p.
7. **Wilde D.** The ALPHA language. — Rennes, 1994. — 23p. — (Rapp./INRIA; No.2295).

В. И. Шелехов

ЯЗЫК ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ Р

Препринт
101

Рукопись поступила в редакцию 15.11.02
Рецензент Ф. А. Мурзин
Редактор З. В. Скок

Подписано в печать 6.12.02
Формат бумаги 60 × 84 1/16 Объем 2.3 уч.-изд.л., 2.6 п.л.
Тираж 50 экз.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6