Язык предикатного программирования P

Д.Ю. Першин, Н.С. Карнаухов, В.И. Шелехов, Д.Р. Янбулатов

Детально описывается синтаксис и семантика языка предикатного программирования. Данное описание предназначено для пользователей и разработчиков системы предикатного программирования.

Изменения версии 0.24

28 мая 2024г.

1. Добавлены средства работы с графами. Введена конструкция ИЗОБРАЖЕНИЕ-ГРАФОВЫХ ТИПОВ, **node**, **nodeset**, **edgeset** для типов вершин и множеств вершин и ребер (см. раздел 9).

2. Конструкции для представления списков для эффективной реализации и оптимизации памяти для строк, перенесены из разделов 7.2 и 7.3 в Приложения, разд. 13.8 и 13.9, соответственно.

3. Раздел 11. Добавлена конструкция *секция* для описания переменных состояний, а также констант и типов автоматной программы.

4. Раздел 7. Уточнены понятия совместимости и согласованности типов.

5. Раздел 7.4. Введена гиперфункция **pick** для недетерминированного выбора элемента из множества.

6. Раздел 6. Добавлена возможность модификации массива по набору индексов, задаваемого произвольным множеством.

1. Введение

Язык предикатного программирования **P** (Predicate programming language) ориентирован главным образом на разработку программ повышенной надежности с применением дедуктивной верификации. Язык **P** является универсальным и может использоваться для построения широкого класса программ-функций [8]. В первую очередь, это программы для задач дискретной и вычислительной математики. Язык **P** также поддерживает автоматное программирование [5, 6, 10] для реализации реактивных систем – программ из класса программ-процессов [8].

Предикатная программа – это предикат (логическая формула) в виде вычислимого оператора. Используется синтаксис языков типа C. Допускаются императивные конструкции. При этом программа всегда приводима к правильному предикату. Такой стиль отличен от монад в языках.

Система типов языка **P** включает: массивы, алгебраические типы (списки, деревья и др.), предикатные типы, подтипы (множество истинности предиката), параметрические типы, графы.

Запрещены циклы и указатели. Вместо циклов используются рекурсивные программы, а вместо указателей – объекты алгебраических типов.

Эффективность предикатных программ достигается применением следующих оптимизирующих трансформаций, переводящих программу на императивное расширение языка P:

- замена хвостовой рекурсии циклом;
- открытая подстановка тела программы на место его вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;
- кодирование алгебраических типов (списков и деревьев) с помощью массивов и указателей.

С помощью трансформаций можно получить программу предельной эффективности.

Язык предикатного программирования P впервые представлен в работе [1]. Полное описание языка P появилось в препринте [2]. Последующие модификации и расширения языка инициированы использованием языка для описания различных алгоритмов [3, 4]. Введены средства спецификации предикатных программ в виде предусловий и постусловий. Язык P расширен для спецификации и реализации реактивных систем, определяемых в виде совокупности взаимодействующих процессов [5, 6, 10]. В языке P появились средства работы с графами.

Первые версии языка P по стилю были ближе к языкам Паскаль и Модула-2. Версия 0.6 [11] по синтаксису, набору операторов и операций и др. особенностям существенно приближена к стилю языков типа C. Определенное влияние на язык P оказал язык спецификаций PVS [7]. Введены алгебраические типы. Последовательности заменены списками.

Синтаксис языка P описывается на расширенном языке Бэкусовских нормальных форм (БНФ) со следующими особенностями:

- терминальные символы выделены жирным шрифтом
- [фрагмент] - означает возможное отсутствие в синтаксическом правиле заключенного в квадратные скобки фрагмента; фрагмент определяет последовательность терминальных и нетерминальных символов
- (фрагмент)* - определяет повторение фрагмента нуль или более раз; круглые скобки могут быть опущены, если фрагмент состоит из одного символа
- (фрагмент)+ - определяет повторение фрагмента один или более раз
- запись вида [:CLASS:] обозначает символ указанного класса; используются следующие классы символов:

alpha	Буквенный символ, принадлежащий латинскому или русскому алфавиту; разрешаются заглавные и строчные буквы
digit	Цифра (0, 1, 2, 3, 4, 5, 6, 7, 8 или 9)
alnum	Символ, принадлежащий alpha или digit
blank	Пробел или символ табуляции
xdigit	Шестнадцатеричная цифра (digit либо заглавная или строчная буква от A до F)

print	Символ, для которого определено начертание (т.е., символ из alnum, blank либо какой-либо другой символ, который можно отобразить)

2. Лексемы

Текст программы представлен в виде последовательности строк символов. Переход на новую строку эквивалентен символу “пробел”. Программа может содержать комментарии, текст которых считается не принадлежащим тексту программы. Комментарий начинается парой символов /* и завершается парой символов */. Второй вид комментариев начинается с пары символов // и продолжается до конца текущей строки текста. Текст программы составляется из лексем следующего вида:

```
ЛЕКСЕМА ::=
        ИДЕНТИФИКАТОР | КЛЮЧЕВОЕ-СЛОВО | КОНСТАНТА |
        ОПЕРАЦИЯ | РАЗДЕЛИТЕЛЬ| МЕТКА
ИДЕНТИФИКАТОР  ::= НАЧАЛО-ИДЕНТИФИКАТОРА СИМВОЛ-ИДЕНТИФИКАТОРА*
НАЧАЛО-ИДЕНТИФИКАТОРА ::= _ | [:alpha:]
СИМВОЛ-ИДЕНТИФИКАТОРА ::= _ | [:alnum:]
```

Идентификатор используется для именования предикатов, переменных, типов и других объектов. Использование специального имени “_” в качестве результирующей переменной вызова предиката (см. разд. 3.3) обозначает пустой результат, неиспользуемый в дальнейшем вычислении. Имя “_” зарезервировано в языке P - его нельзя использовать для именования объектов программы. Другие идентификаторы, начинающиеся с “_”, считаются обычными и могут использоваться в описаниях программы.

```
МЕТКА    ::= СИМВОЛ-ИДЕНТИФИКАТОРА*

КЛЮЧЕВОЕ-СЛОВО ::=
  after | array | as | axiom | bool | break | case | char | class | context
 |edgeset |else |elsif | enum | exists | extends | default | false | for | forall
 |formula | if | in | inf | int | invariant | inv | import | lemma | message |
measure
 | module | nan | nat | new | nil | node | nodeset | or | post | pre | process
 |predicate | real | receive | send | set | spawn| string | struct | subtype | switch
 | time |theorem | theory |type | true | union | var | while | with | xor

ОПЕРАЦИЯ ::=
       + | - | * | / | % | ^ | ! | << | >> | ~ | & | | | ? | = | < | > | <=
      | >= | != | => | <=>

РАЗДЕЛИТЕЛЬ ::=    ( | ) | [ | ] | { | } | , | ; | : | . | .. | [:blank:]

КОНСТАНТА ::=
     ЦЕЛАЯ-КОНСТАНТА
   | ВЕЩЕСТВЕННАЯ-КОНСТАНТА
   | СИМВОЛЬНАЯ-КОНСТАНТА
   | СТРОКОВАЯ-КОНСТАНТА
   | ЛОГИЧЕСКАЯ-КОНСТАНТА
   | КОНСТАНТА-ПУСТО
```

```
ЦЕЛАЯ-КОНСТАНТА   ::= ЦЕЛОЕ-ЧИСЛО
ЦЕЛОЕ-ЧИСЛО       ::= [ЗНАК] ЦИФРЫ | 0x ШЕСТНАДЦАТЕРИЧНЫЕ-ЦИФРЫ
ЗНАК              ::= + | -
ЦИФРЫ             ::= ЦИФРА+
ЦИФРА             ::= [:digit:]
ШЕСТНАДЦАТЕРИЧНЫЕ-ЦИФРЫ ::= ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА+
ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА ::= [:xdigit:]
ВЕЩЕСТВЕННАЯ-КОНСТАНТА ::=
    [ЗНАК] ЦИФРЫ [. ЦИФРЫ] [ПОРЯДОК]
  | [ЗНАК] inf
  | nan
ПОРЯДОК ::= (e | E) [ЗНАК] ЦИФРЫ
```

Для вещественных типов (см. разд. 7) определены специальные значения: **inf** (бесконечность) и **nan** (не число). Значение **nan** возникает в процессе вычисления как результат взятия квадратного корня из отрицательного числа, деления ноля на ноль, умножения ноля на бесконечность и других операций над вещественными числами, когда результат не может быть определён.

```
СИМВОЛЬНАЯ-КОНСТАНТА : := ' СИМВОЛ '
СИМВОЛ :  := ОБЫЧНЫЙ-СИМВОЛ | СПЕЦИАЛЬНЫЙ-СИМВОЛ |
             ШЕСТНАДЦАТЕРИЧНЫЙ-КОД-СИМВОЛА
ОБЫЧНЫЙ-СИМВОЛ    ::= любой символ из [:print:], кроме ' и "
СПЕЦИАЛЬНЫЙ-СИМВОЛ  ::= \" | \' | \\ | \0 | \n | \r | \t
ШЕСТНАДЦАТЕРИЧНЫЙ-КОД-СИМВОЛА ::=
  \x ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА
     [ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА [ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА
                               [ШЕСТНАДЦАТЕРИЧНАЯ-ЦИФРА]]]
```

Константа типа **char** (см. разд. 7) может быть либо символом класса **print** (см. разд. 1), либо специальной комбинацией символов, заключённой в одинарные кавычки. Специальные комбинации символов могут также встречаться в составе строковых констант.

\"	Двойные кавычки
\'	Одинарные кавычки
****	Обратная косая черта
\0	Символ с кодом 0
\n	Перевод строки (код 10)
\r	Возврат каретки (код 13)
\xNNNN	Символ с шестнадцатеричным кодом NNNN, состоящим не более чем из четырёх цифр.

Таблица 1 Специальные комбинации символов

```
СТРОКОВАЯ-КОНСТАНТА   ::= " СИМВОЛ * "
ЛОГИЧЕСКАЯ-КОНСТАНТА ::= true | false
КОНСТАНТА-ПУСТО       ::= nil
```

3. Предикатные программы

Полная предикатная программа состоит из набора предикатных программ в виде определений предикатов.

3.1. Определение предиката

```
ОПРЕДЕЛЕНИЕ-ПРЕДИКАТА ::= ИМЯ-ПРЕДИКАТА ОПИСАНИЕ-ПРЕДИКАТА
ИМЯ-ПРЕДИКАТА   ::= ИДЕНТИФИКАТОР
```

Значением ОПИСАНИЯ-ПРЕДИКАТА является предикат, обозначаемый именем предиката.

```
ОПИСАНИЕ-ПРЕДИКАТА ::= ОПИСАНИЕ-ПРЕДИКАТА-ФУНКЦИИ |
                       ОПИСАНИЕ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ
ОПИСАНИЕ-ПРЕДИКАТА-ФУНКЦИИ ::=
    ЗАГОЛОВОК-ПРЕДИКАТА [pre ПРЕДУСЛОВИЕ] [post ПОСТУСЛОВИЕ]
    [measure ВЫРАЖЕНИЕ] ТЕЛО-ПРЕДИКАТА
ЗАГОЛОВОК-ПРЕДИКАТА  ::=
     [ОПИСАНИЯ-ВНЕШНИХ-АРГУМЕНТОВ] ( [ОПИСАНИЯ-АРГУМЕНТОВ]: ОПИСАНИЯ-
РЕЗУЛЬТАТОВ )

ТЕЛО-ПРЕДИКАТА  ::= БЛОК
ПРЕДУСЛОВИЕ     ::= ФОРМУЛА
ПОСТУСЛОВИЕ     ::= ФОРМУЛА
```

Значения аргументов предиката должны удовлетворять предусловию. По завершении исполнения тела предиката должно быть истинным постусловие, связывающее значения аргументов и результатов. Мера используется для дедуктивной верификации рекурсивного предиката.

```
ОПИСАНИЯ-РЕЗУЛЬТАТОВ ::=
        ИЗОБРАЖЕНИЕ-ТИПА ИМЯ-РЕЗУЛЬТАТА (, ИМЯ-РЕЗУЛЬТАТА)*
        [, ОПИСАНИЯ-РЕЗУЛЬТАТОВ]
ИМЯ-РЕЗУЛЬТАТА    ::= ИДЕНТИФИКАТОР | ИДЕНТИФИКАТОР ’
```

Имя вида имя’ используется для именования результирующего параметра, при условии, что имя описано в качестве одного из аргументов с тем же типом. В императивной программе, получаемой трансформацией предикатной программы, результат с именем имя’ склеивается с соответствующим аргументом имя. Если тип аргумента имя параметризован (см. разд. 7), то значение параметра для типа результата имя’ может отличаться.

```
ОПИСАНИЯ-ГРУППЫ-АРГУМЕНТОВ  ::=
        ИЗОБРАЖЕНИЕ-ТИПА ИМЯ-АРГУМЕНТА (, ИМЯ-АРГУМЕНТА)* |
        ИЗОБРАЖЕНИЕ-ТИПА-КАК-ПАРАМЕТРА
ОПИСАНИЯ-АРГУМЕНТОВ  ::=
        ОПИСАНИЯ-ГРУППЫ-АРГУМЕНТОВ [, ОПИСАНИЯ-АРГУМЕНТОВ]
ИМЯ-АРГУМЕНТА  ::= ИДЕНТИФИКАТОР
```

```
assign (int from : int to) { to = from }
main (list (string) argv : int ret_code) { ret_code = 0 }
power (real x, int p : real x’) { x’ = x^p }
```

Примеры определений предикатов

```
ОПИСНИЕ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ ::=
        ЗАГОЛОВОК-ПРЕДИКАТА-ГИПЕРФУНКЦИИ [ПРЕДУСЛОВИЯ-ГИПЕРФУНКЦИИ]
        [ПОСТУСЛОВИЯ-ВЕТВЕЙ] ТЕЛО-ПРЕДИКАТА
ЗАГОЛОВОК-ПРЕДИКАТА-ГИПЕРФУНКЦИИ   ::=
        [ОПИСАНИЯ-ВНЕШНИХ-АРГУМЕНТОВ]
        ( [ОПИСАНИЯ-АРГУМЕНТОВ] : ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ГИПЕРФУНКЦИИ )
ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ГИПЕРФУНКЦИИ  ::=
        [ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ВЕТВИ] [# МЕТКА-ВЕТВИ]
        (:  [ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ВЕТВИ] [# МЕТКА-ВЕТВИ])+
ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ВЕТВИ ::= ОПИСАНИЯ-РЕЗУЛЬТАТОВ
МЕТКА-ВЕТВИ ::= МЕТКА
```

После двоеточия описываются параметры-результаты очередной ветви гиперфункции. Ветви гиперфункции именуются метками. При отсутствии метки при описании результатов ветви, ветвь именуется целым - порядковым номером ветви начиная с 1. Исполнение гиперфункции завершается выбором одной из ветвей и вычислением значений результатов этой ветви. При этом результаты других ветвей не определены. Оператор перехода #имяВетви в теле предиката завершает исполнение гиперфункции ветвью с именем имяВетви.

```
ПРЕДУСЛОВИЯ-ГИПЕРФУНКЦИИ ::=
            [ОБЩЕЕ-ПРЕДУСЛОВИЕ] (ПРЕДУСЛОВИЕ-ВЕТВИ)+
ОБЩЕЕ-ПРЕДУСЛОВИЕ ::= pre ПРЕДУСЛОВИЕ
ПРЕДУСЛОВИЕ-ВЕТВИ ::= pre МЕТКА-ВЕТВИ : ПРЕДУСЛОВИЕ
```

Отсутствие общего предусловия определяет допустимость произвольных значений аргументов в соответствии с их типами. Предусловие ветви определяет дополнительное условие на значения аргументов, при котором исполнение гиперфункции завершается выбором этой ветви. При этом значения аргументов должны удовлетворять общему предусловию. Предусловие последней ветви обычно не указывается, поскольку оно получается дополнением предусловий предыдущих ветвей.

```
ПОСТУСЛОВИЯ-ВЕТВЕЙ    ::= (ПОСТУСЛОВИЕ-ВЕТВИ)+
ПОСТУСЛОВИЕ-ВЕТВИ     ::= post МЕТКА-ВЕТВИ : ПОСТУСЛОВИЕ
```

Постусловие ветви связывает значения аргументов и результатов ветви. Ветвь без результатов постусловия не имеет.

```
ОПИСАНИЯ-ВНЕШНИХ-АРГУМЕНТОВ  ::= ( ОПИСАНИЯ-АРГУМЕНТОВ )
```

Внешние аргументы являются начальной частью списка аргументов программы. Механизм внешних аргументов в стиле каррирования способствует улучшению восприятия программы. Вызов программы при наличии внешних аргументов определен в разд. 3.3. Ограничение: внешние аргументы недопустимы для программы, единственным результатом которой является переменная предикатного типа.

3.2. Спецификация предиката

Предикат, используемый в программе, должен иметь определение предиката. Если предикат определяется в другом модуле программы, то предикат может быть представлен своей спецификацией.

```
СПЕЦИФИКАЦИЯ-ПРЕДИКАТА ::=
        ИМЯ-ПРЕДИКАТА   ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА
```

Имя предиката обозначает предикат, представленный описанием спецификации предиката.

```
ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА ::=
      ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА-ФУНКЦИИ |
      ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ
ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА-ФУНКЦИИ ::=
      ЗАГОЛОВОК-ПРЕДИКАТА [pre ПРЕДУСЛОВИЕ][post ПОСТУСЛОВИЕ]
ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ ::=
      ЗАГОЛОВОК-ПРЕДИКАТА-ГИПЕРФУНКЦИИ [ПРЕДУСЛОВИЯ-ГИПЕРФУНКЦИИ]
      [ПОСТУСЛОВИЯ-ВЕТВЕЙ]
```

3.3. Вызов предиката

Элементарным оператором программы является вызов предиката.

```
ВЫЗОВ-ПРЕДИКАТА ::=
     ВЫЗОВ-ПРЕДИКАТА-ФУНКЦИИ | ВЫЗОВ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ
ВЫЗОВ-ПРЕДИКАТА-ФУНКЦИИ ::=
      ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА [ВНЕШНИЕ-АРГУМЕНТЫ]
      ( [АРГУМЕНТЫ] : РЕЗУЛЬТАТЫ )
ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА ::=
    [ИМЯ-МОДУЛЯ .] ИМЯ-ПРЕДИКАТА | [ОБЪЕКТ-КЛАССА .] ИМЯ-ПРЕДИКАТА |
    ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ
```

ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ должно быть предикатного типа; его значением является предикат, запускаемый на исполнение данным вызовом.

```
ОБЪЕКТ-КЛАССА ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ
```

Доступ к предикату, являющемуся методом класса, реализуется через объект класса.

```
АРГУМЕНТЫ ::= СПИСОК-ВЫРАЖЕНИЙ
СПИСОК-ВЫРАЖЕНИЙ ::= ВЫРАЖЕНИЕ [, СПИСОК-ВЫРАЖЕНИЙ]
```

Типы аргументов вызова должны быть совместимы (см. разд. 7) с типами соответствующих аргументов определения (или спецификации) вызываемого предиката.

```
РЕЗУЛЬТАТЫ  ::=  ПЕРЕМЕННАЯ [, РЕЗУЛЬТАТЫ] |
                 РЕЗУЛЬТАТ-ЛОКАЛ [, РЕЗУЛЬТАТЫ]
РЕЗУЛЬТАТ-ЛОКАЛ   ::=   ОПИСАНИЕ-ПЕРЕМЕННОЙ
ОПИСАНИЕ-ПЕРЕМЕННОЙ ::=
                 ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ   ИДЕНТИФИКАТОР
ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ  ::=
                 ИЗОБРАЖЕНИЕ-ТИПА [:blank:]| var [:blank:]
```

Использование **var** возможно вместо соответствующего ИЗОБРАЖЕНИЕ-ТИПА, поскольку тип результата-локала в большинстве случаев можно восстановить по определению вызываемого предиката. Описатель **var** также может использоваться при описании локальной переменной в случае, когда тип этой переменной легко определяется из контекста дальнейшего использования переменной (см. разд. 4, 5).

Типы результатов вызова должны совпадать с типами соответствующих результатов в определении вызываемого предиката. Переменная, представленная описанием результата-локала, определяется как локальная в теле предиката, содержащем данный вызов. Описание локальной результирующей переменной внутри вызова эквивалентно описанию этой переменной перед вызовом.

Использование специального имени “_” в качестве результирующей переменной обозначает пустой результат: результат по этой позиции, получаемый при завершении исполнения вызова предиката, не используется в дальнейшем исполнении. Имя “_” зарезервировано в языке Р - его нельзя использовать для именования объектов программы.

```
ВЫЗОВ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ ::=
        ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА [ВНЕШНИЕ-АРГУМЕНТЫ]
        ( [АРГУМЕНТЫ] (: РЕЗУЛЬТАТЫ-ВЕТВИ)+ )
РЕЗУЛЬТАТЫ-ВЕТВИ  ::= [РЕЗУЛЬТАТЫ] [ОПЕРАТОР-ПЕРЕХОДА]
ОПЕРАТОР-ПЕРЕХОДА ::= # МЕТКА
```

В качестве метки в операторе указывается либо метка ветви предиката, в теле которого находится данный вызов, либо метка ветви обработчика; подробнее см. в разд. 5.

Исполнение вызова предиката реализуется следующим образом. 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 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. Если в данной ветви вызова указан оператор перехода, то либо происходит переход на локальную метку, либо завершается тело предиката, содержащего вызов, той ветвью, которая указана в операторе перехода. Отметим, что подстановка аргументов и результатов предиката реализуется по значению.

```
ВЫЗОВ-ФУНКЦИИ ::=
        ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА [ВНЕШНИЕ-АРГУМЕНТЫ] ( [АРГУМЕНТЫ] )
```

Вызов предиката-функции может иметь вид вызова функции. Результатом исполнения является набор значений результирующих переменных исполненного определения предиката.

```
real r;
sqrt (2 : r);
real q = sqrt (3);

Comp (list(int) s: : int d, list (int) r) // спецификация гиперфункции Comp
pre  1: s = nil
post 2: s = cons(d, r) ;
/* если список s - пуст, реализуется первая ветвь гиперфункции, иначе реализуется
   вторая ветвь с результатами: d - первый элемент, r - хвост списка */

elemTwo (list(int) s: int e #value : #empty)
pre  empty: s = nil or s.cdr = nil
post value: e = (s.cdr).car;
{   Comp (s  : #empty : int e1, list(int) s1 #ok)
          case ok: Comp (s1 : #empty : e, list(int) s2 #value);
}
/* гиперфункция elemTwo извлекает второй элемент списка, если элемент существует  */
```

Примеры вызова предикатов, определения, спецификации и вызова гиперфункций

В приведенном выше определении предиката elemTwo переход по локальной метке ok является излишним. Кроме того, результаты e1 и s2 далее нигде не используются. Определение предиката elemTwo, представленное ниже, исправляет отмеченные недостатки:

```
elemTwo (list(int) s: int e #value : #empty)
pre 1: s = nil or s.cdr = nil
post value: e = s.cdr.car;
{    Comp (s  : #empty : _, list(int) s1 );
     Comp (s1 : #empty : e, _ #value);
}
```

```
ВНЕШНИЕ-АРГУМЕНТЫ ::= АРГУМЕНТЫ
```

Вызов программы H с набором внешних аргументов x , набором аргументов y и набором результатов z записывается в виде H(x)(y**:** z). Вызов может быть записан в виде H(y**:** z), если переменные набора x были указаны в составе ОПИСАНИЯ-КОНТЕКСТА (см. разд. 4). Внешние аргументы являются начальной частью списка аргументов программы.

Исполнение вызова H(x)(y**:** z) начинается с вычисления значений внешних аргументов x, затем аргументов y. Дальнейшее исполнение реализуется также как для вызова без внешних аргументов. Вычисление аргументов в полном наборе x и y может проводиться независимо, т.е. параллельно.

4. Программа

Программа состоит из одного или нескольких модулей. Модуль определяет независимую часть программы.

```
ОПИСАНИЕ-МОДУЛЯ ::= [ЗАГОЛОВОК-МОДУЛЯ] ОПИСАНИЯ-МОДУЛЯ
ЗАГОЛОВОК-МОДУЛЯ ::= module ИМЯ-МОДУЛЯ  [ ( ОПИСАНИЯ-АРГУМЕНТОВ ) ];
ИМЯ-МОДУЛЯ ::= ИДЕНТИФИКАТОР
```

Аргументы, описанные в круглых скобках, являются формальными параметрами модуля. Структура ОПИСАНИЙ-АРГУМЕНТОВ определена в разд. 3.1.

```
ОПИСАНИЯ-МОДУЛЯ ::= ОПИСАНИЕ-МОДУЛЯ [ ; ОПИСАНИЯ-МОДУЛЯ ]
ОПИСАНИЕ-МОДУЛЯ ::= ИМПОРТ-МОДУЛЯ | ОПИСАНИЕ
ОПИСАНИЕ ::=
   ОПИСАНИЕ-ТИПА |
   ОПИСАНИЕ-ПЕРЕМЕННЫХ |
   ОПРЕДЕЛЕНИЕ-ПРЕДИКАТА |
   СПЕЦИФИКАЦИЯ-ПРЕДИКАТА |
   ОПИСАНИЕ-КОНТЕКСТА |
   ОПИСАНИЕ-ФОРМУЛЫ |
   ОПИСАНИЕ-ТЕОРИИ |
   ОПРЕДЕЛЕНИЕ-КЛАССА |
   ОПИСАНИЕ-СООБЩЕНИЯ |
   ОПРЕДЕЛЕНИЕ-ПРОЦЕССА
```

ОПИСАНИЕ-ФОРМУЛЫ (см. разд. 10) определяет функцию или предикат; они встречаются в предусловиях, постусловиях и формулах. ОПИСАНИЕ-ТЕОРИИ определяет набор лемм (теорем) и аксиом (см. разд. 10).

ОПРЕДЕЛЕНИЕ-КЛАССА описано в разд. 6.5. Две последних альтернативы ОПИСАНИЯ используются для задания процессов; см. разд. 11.

```
ИМПОРТ-МОДУЛЯ ::=
        import ИМЯ-МОДУЛЯ [ ( АРГУМЕНТЫ )] [ as ЛОКАЛЬНОЕ-ИМЯ-МОДУЛЯ ]
ЛОКАЛЬНОЕ-ИМЯ-МОДУЛЯ  ::=  ИДЕНТИФИКАТОР
```

Имена модулей, встречающиеся в описаниях данного модуля, должны быть определены конструкцией ИМПОРТ-МОДУЛЯ. Если импортируемый модуль имеет формальные параметры, то в круглых скобках задается соответствующий набор фактических параметров. Конструкция АРГУМЕНТЫ определена в разд. 3.3. Правила подстановки фактических параметров на место формальных те же самые, что и для вызова предиката. ЛОКАЛЬНОЕ-ИМЯ-МОДУЛЯ используется в теле модуля как эквивалент
ИМЯ-МОДУЛЯ (АРГУМЕНТЫ).

Описания переменных как часть ОПИСАНИЙ-МОДУЛЯ определяют глобальные переменные, являющиеся внешними параметрами задачи. В определениях предикатов эти переменные считаются аргументами. Они используются в теле предиката, но не упоминаются в заголовке.

```
ОПИСАНИЕ-ПЕРЕМЕННЫХ ::=
    ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ
    ИМЯ-ПЕРЕМЕННОЙ (, ИМЯ-ПЕРЕМЕННОЙ)*
ОПИСАНИЕ-ПЕРЕМЕННЫХ-С-ИНИЦИАЛИЗАЦИЕЙ ::=
    ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ
    ОПРЕДЕЛЕНИЕ-ПЕРЕМЕННОЙ (, ОПРЕДЕЛЕНИЕ-ПЕРЕМЕННОЙ)*

ОПРЕДЕЛЕНИЕ-ПЕРЕМЕННОЙ  ::=
 ИМЯ-ПЕРЕМЕННОЙ = ВЫРАЖЕНИЕ | ИМЯ-ПЕРЕМЕННОЙ
```

Тип выражения должен быть совместим с типом переменой (см. разд. 7), которой присваивается значение переменной.

```
ОПИСАНИЕ-КОНТЕКСТА ::= context ОПИСАНИЯ-АРГУМЕНТОВ
```

ОПИСАНИЕ-КОНТЕКСТА декларирует, что в последующих описаниях текущего модуля перечисленные переменные могут быть опущены при использовании их в качестве параметров типов, а также в вызовах программ и формул для внешних аргументов. Это способствует улучшению восприятия программы.

Модуль определяет совокупность имен и обозначаемых ими объектов: предикатов, типов, переменных, полей структур, конструкторов и распознавателей объединений, процессов, классов и сообщений. Объекты, представленные описаниями модуля, определяют область глобальных имен. Каждое описание предиката определяет автономную систему локализации имен: параметров, меток и локальных переменных. В описании предиката одно имя не может использоваться для двух разных объектов. Если глобальное имя используется в определении предиката, то это имя не может определяться в качестве имени метки, локальной переменной или параметра.

Использованию имени в тексте программы должно предшествовать его описанию. Исключение для МЕТКИ и ИМЕНИ-ПРЕДИКАТА. Вхождения ИМЕНИ-ПРЕДИКАТА и ВЫЗОВА-ПРЕДИКАТА могут предшествовать соответствующему ОПРЕДЕЛЕНИЮ-ПРЕДИКАТА. Однако для ВЫЗОВА-ФУНКЦИИ ее спецификация должна быть предварительно описана.

Следующей областью локализации являются конструкции: изображение подтипа, определение массива, элемент агрегата вида “итерация выражений”. Переменные, определенные в каждой из этих конструкций, локализованы в конструкции и недоступны вне нее. Класс также определяет независимую область локализации. Поля класса доступны префиксацией объекта класса (см. разд. 6).

Исполнение программы начинается исполнением предиката с именем main. Рекомендован предикат со следующими параметрами: main (list(string) argv : int ret_code). Параметр argv поставляет массив входных аргументов при исполнении программы. Параметру ret_code присваивается значение кода выхода.

Для вывода информации используется оператор print, аргументы которого – выражения и массивы.

5. Операторы

```
БЛОК      ::= { ОПЕРАТОР }
ОПЕРАТОР  ::=  [ОПИСАНИЕ-ПЕРЕМЕННЫХ ;] ЗАМКНУТЫЙ-ОПЕРАТОР |
               [ОПИСАНИЕ-ПЕРЕМЕННЫХ ;] ПАРАЛЛЕЛЬНЫЙ-ОПЕРАТОР |
               [ОПИСАНИЕ-ПЕРЕМЕННЫХ ;] ОПЕРАТОР-СУПЕРПОЗИЦИИ
```

Описанные переменные считаются локальными в теле предиката. Тип локальной переменной может быть также определен описателем **var** (см. разд. 3.3) в случае, когда тип легко определяется из контекста дальнейшего использования переменной.

```
ЗАМКНУТЫЙ-ОПЕРАТОР ::=
        ОПЕРАТОР-ПРИСВАИВАНИЯ |
        ОПЕРАТОР-МУЛЬТИ-ПРИСВАИВАНИЯ |
        ВЫЗОВ-ПРЕДИКАТА-ФУНКЦИИ |
        ВЫЗОВ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ [ОПЕРАТОР-ОБРАБОТКИ-ВЕТВЕЙ] |
        УСЛОВНЫЙ-ОПЕРАТОР |
        БЛОК [ОПЕРАТОР-ОБРАБОТКИ-ВЕТВЕЙ] |
        ОПЕРАТОР-ВЫБОРА |
        ОПРЕДЕЛЕНИЕ-ЛОКАЛЬНОГО-ПРЕДИКАТА
ОПЕРАТОР-ПРИСВАИВАНИЯ ::= ПЕРЕМЕННАЯ = ВЫРАЖЕНИЕ
```

В результате исполнения оператора присваивания переменной присваивается значение выражения. Типы выражения и переменной должны быть совместимы (см. разд. 7).

```
ОПЕРАТОР-МУЛЬТИ-ПРИСВАИВАНИЯ ::=
        МУЛЬТИ-ПЕРЕМЕННАЯ = МУЛЬТИ-ВЫРАЖЕНИЕ
МУЛЬТИ-ПЕРЕМЕННАЯ  ::= | СПИСОК-ПЕРЕМЕННЫХ | |
                         СПИСОК-ПЕРЕМЕННЫХ
СПИСОК-ПЕРЕМЕННЫХ  ::= ПЕРЕМЕННАЯ [, СПИСОК-ПЕРЕМЕННЫХ ]
```

Два разных способа представления мульти-переменной считаются равноценными.

Мульти-переменная определяет упорядоченный набор переменных. Число переменных в наборе должно быть больше 1.

```
МУЛЬТИ-ВЫРАЖЕНИЕ ::= | СПИСОК-ВЫРАЖЕНИЙ | |
                      СПИСОК-ВЫРАЖЕНИЙ
```

Значением мульти-выражения является набор значений, получающихся в результате вычисления перечисленных выражений. Выражения вычисляются независимо, т.е. параллельно. Полученный набор значений одновременно (синхронно) присваивается соответствующим переменным. Типы переменных в

мульти-переменной должны быть совместимы (см. разд. 7) с типами соответствующих выражений из мульти-выражения.

```
ОПЕРАТОР-ЕСЛИ  ::=
    if ( ВЫРАЖЕНИЕ ) ЗАМКНУТЫЙ-ОПЕРАТОР [ОПЕРАТОР-ПЕРЕХОДА]

УСЛОВНЫЙ-ОПЕРАТОР ::=
    ОПЕРАТОР-ЕСЛИ
    ( elsif ( ВЫРАЖЕНИЕ ) ЗАМКНУТЫЙ-ОПЕРАТОР [ОПЕРАТОР-ПЕРЕХОДА] )*
    else ЗАМКНУТЫЙ-ОПЕРАТОР [ОПЕРАТОР-ПЕРЕХОДА]
```

В операторе перехода указывается либо метка ветви гиперфункции, либо метка ветви обработчика, следующего за блоком, содержащем данный условный оператор.

```
if (x > 0) s = 1
elsif (x < 0) s = -1
else  s = 0
```

Пример использования условного оператора

```
ПАРАЛЛЕЛЬНЫЙ-ОПЕРАТОР ::=
      ЗАМКНУТЫЙ-ОПЕРАТОР (|| ЗАМКНУТЫЙ-ОПЕРАТОР)+
```

Операторы, образующие параллельный оператор, должны иметь непересекающиеся наборы результирующих переменных. Исполнение параллельного оператора слагается из исполнения входящих в него операторов. Операторы выполняются независимо друг от друга; они могут исполняться параллельно.

```
ОПЕРАТОР-СУПЕРПОЗИЦИИ ::=  ОПЕРАТОР (; ОПЕРАТОР)+
```

В цепочке операторов, составляющих оператор суперпозиции, 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.

```
ЗАГОЛОВОК-ОПЕРАТОРА-ВЫБОРА ::= switch ( ВЫРАЖЕНИЕ )
ОПЕРАТОР-ВЫБОРА  ::=
     ЗАГОЛОВОК-ОПЕРАТОРА-ВЫБОРА
                  { ОПЕРАТОР-АЛЬТЕРНАТИВЫ+
                    [default : ОПЕРАТОР [ОПЕРАТОР-ПЕРЕХОДА]]
                  }
ОПЕРАТОР-АЛЬТЕРНАТИВЫ  ::=
    case АЛЬТЕРНАТИВА (, АЛЬТЕРНАТИВА)* : ОПЕРАТОР
    [ОПЕРАТОР-ПЕРЕХОДА]
АЛЬТЕРНАТИВА    ::= ВЫРАЖЕНИЕ | ОПРЕДЕЛЕНИЕ-КОНСТРУКТОРА
```

Выполняется тот оператор альтернативы, для которого значение АЛЬТЕРНАТИВЫ совпадает со значением выражения в заголовке оператора выбора. Оператор после **default** выполняется в случае, когда нет ОПЕРАТОРА-АЛЬТЕРНАТИВЫ с требуемым значением АЛЬТЕРНАТИВЫ. Трактовка операторов перехода та же, что и для условного оператора.

Если в позиции выражения в заголовке оператора выбора находится переменная типа объединения, то в качестве АЛЬТЕРНАТИВЫ указывается ОПРЕДЕЛЕНИЕ-КОНСТРУКТОРА. Особенности выполнения ОПЕРАТОРА-АЛЬТЕРНАТИВЫ определены в разд. 7.

```
ОПЕРАТОР-ОБРАБОТКИ-ВЕТВЕЙ  ::=
        (case [МЕТКА-ВЕТВИ-ОБРАБОТЧИКА :] ОПЕРАТОР)+
МЕТКА-ВЕТВИ-ОБРАБОТЧИКА  ::=  МЕТКА
```

Оператор обработки ветвей следует за вызовом гиперфункции либо за блоком, содержащим операторы перехода. Если предыдущий оператор нормально завершается, то оператор обработки ветвей игнорируется. Нормальное завершение возможно и для вызова гиперфункции, когда на одной из ветвей гиперфункции опущена метка перехода, что означает суперпозицию (или параллельную композицию) с последующим оператором, а на другой ветви имеется переход на метку ветви обработчика или на метку ветви гиперфункции, в теле которой находится данный оператор.

Исполнение оператора обработки ветвей инициируется оператором перехода, находящемся в предыдущем операторе. При этом исполняется оператор, помеченный соответствующей меткой, после чего исполнение оператора обработки ветвей и предыдущего оператора считается завершившимся.

В вызове гиперфункции могут быть опущены переходы, а в последующем операторе обработки ветвей - метки ветвей обработчика. В этом случае порядок ветвей обработчика соответствует порядку ветвей вызова гиперфункции.

```
A(i: y #1: #2)
    case 1: B(y, i: u)
    case 2: D(i: u);
```

Пример вызова гиперфункции с обработчиком ветвей

```
ОПРЕДЕЛЕНИЕ-ЛОКАЛЬНОГО-ПРЕДИКАТА  ::= ОПРЕДЕЛЕНИЕ-ПРЕДИКАТА
```

Определение локального предиката допускает использование параметров и локальных переменных предиката, в теле которого это определение находится. Локальный предикат доступен лишь в теле предиката, внутри которого он описан.

6. Выражения

```
ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ ::=
    ПЕРЕМЕННАЯ |
    ВЫЗОВ-ФУНКЦИИ |
    ( ВЫРАЖЕНИЕ ) |
    ПОЛЕ-ОБЪЕДИНЕНИЯ
```

```
ВТОРИЧНОЕ-ВЫРАЖЕНИЕ ::=
    ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ |
    КОНСТАНТА |
    АГРЕГАТ |
    ИМЯ-ПРЕДИКАТА |
    ГЕНЕРАТОР-ПРЕДИКАТА |
    ИМЯ-ТИПА |
    МОДИФИКАЦИЯ |
    ЭЛЕМЕНТ-СПИСКА |
    ОПРЕДЕЛЕНИЕ-МАССИВА |
    МЕТОД |
    СООБЩЕНИЕ  |
    КОНСТРУКТОР |
    РАСПОЗНАВАТЕЛЬ |
    ВЫРЕЗКА-СПИСКА
```

Имя типа может быть аргументом вызова предиката и изображения типа. ЭЛЕМЕНТ-СПИСКА и ВЫРЕЗКА-СПИСКА определены в разд. 7. Операция ОПРЕДЕЛЕНИЕ-МАССИВА описывается в разд. 8.3. КОНСТРУКТОР, РАСПОЗНАВАТЕЛЬ и ПОЛЕ-ОБЪЕДИНЕНИЯ относятся к объектам типа объединения и описаны в разд. 7. Использование СООБЩЕНИЯ в качестве значения первичного выражения определено в разд.10.

```
ПЕРЕМЕННАЯ ::=  ИМЯ-ПЕРЕМЕННОЙ |
                ЭЛЕМЕНТ-МАССИВА |
                ПОЛЕ-СТРУКТУРЫ |
                ПЕРЕМЕННАЯ-КЛАССА |
                ВЫРЕЗКА-МАССИВА
```

ВЫРЕЗКА-МАССИВА определена в разд. 8.2.

```
ИМЯ-ПЕРЕМЕННОЙ  ::= [ИМЯ-МОДУЛЯ .] ИДЕНТИФИКАТОР [']
```

Переменная вида имя’ используется для именования результата предиката. Переменная имя’ склеивается с переменной имя, являющейся аргументом предиката (см. разд. 3.1).

```
ЭЛЕМЕНТ-МАССИВА   ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ [ ИНДЕКСЫ-МАССИВА ]
ИНДЕКСЫ-МАССИВА   ::= СПИСОК-ВЫРАЖЕНИЙ
```

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа “массив”. Значения ИНДЕКСОВ-МАССИВА должны принадлежать типам индексов соответствующего типа массива.

```
ПОЛЕ-СТРУКТУРЫ   ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ . ИМЯ-ПОЛЯ
```

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа “структура”.

```
ИМЯ-ПОЛЯ    ::= ИДЕНТИФИКАТОР
ПЕРЕМЕННАЯ-КЛАССА ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ . ИМЯ-ПОЛЯ
```

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа “класс”.

```
АГРЕГАТ ::=  АГРЕГАТ-СТРУКТУРА | АГРЕГАТ-МАССИВ | АГРЕГАТ-МНОЖЕСТВО |
             АГРЕГАТ-СПИСОК
```

Агрегат определяет значение структурного типа. Агрегат-массив определяет массив, агрегат-множество - подмножество некоторого множества - значение типа **set** (см. разд. 7.4), агрегат-структура - структуру (как набор полей), агрегат-список - список, заданный последовательностью элементов.

АГРЕГАТ-СТРУКТУРА ::= [ИЗОБРАЖЕНИЕ-ТИПА] **(** ЭЛЕМЕНТЫ-АГРЕГАТА **)**
АГРЕГАТ-МАССИВ ::= [ИЗОБРАЖЕНИЕ-ТИПА] **[** [ЭЛЕМЕНТЫ-АГРЕГАТА] **]**
АГРЕГАТ-МНОЖЕСТВО ::= [ИЗОБРАЖЕНИЕ-ТИПА] **{** [ЭЛЕМЕНТЫ-АГРЕГАТА] **}**
АГРЕГАТ-СПИСОК ::= [ИЗОБРАЖЕНИЕ-ТИПА] **[[** ЭЛЕМЕНТЫ-АГРЕГАТА **]]**

Агрегат **[ ]** обозначает пустой массив (когда верхняя граница индексов меньше нижней). Агрегат **{ }** определяет пустое множество. Если пара скобок **[[** или **]]** встречаются рядом при изображении АГРЕГАТ-МАССИВА, то соседние скобки должны быть разделены пробелом.

ЭЛЕМЕНТЫ-АГРЕГАТА ::= ЭЛЕМЕНТ-АГРЕГАТА [**,** ЭЛЕМЕНТЫ-АГРЕГАТА]

Значением агрегата является набор значений элементов агрегата, перечисленных в скобках. Агрегат может иметь иерархическую структуру, поскольку элементом агрегата может быть агрегат. Тип агрегата определяется типом позиции, в которой находится агрегат. Значение агрегата должно соответствовать типу позиции. Тип агрегата может быть указан явно ИЗОБРАЖЕНИЕМ-ТИПА - это не является необходимым, но может улучшить восприятие программы.

ЭЛЕМЕНТ-АГРЕГАТА ::= [ИНДЕКС-ЭЛЕМЕНТА-АГРЕГАТА **:**] ВЫРАЖЕНИЕ
ИНДЕКС-ЭЛЕМЕНТА-АГРЕГАТА ::= ВЫРАЖЕНИЕ | ИМЯ-ПОЛЯ

Индексы элементов, если присутствуют, должны соответствовать индексам элементов массива или именам полей структуры.

```
type Ar4 = array (real, 1..4);
Ar4 x = [ 0, 1, 2, 3];
```

Пример задания агрегата

ГЕНЕРАТОР-ПРЕДИКАТА ::= **predicate** ОПИСАНИЕ-ПРЕДИКАТА

Исполнение генератора предиката создает новый предикат, операторная часть которого определяется телом предиката, находящегося в составе ОПИСАНИЯ-ПРЕДИКАТА; см. разд. 3.1. Новый предикат становится значением генератора предиката.

МЕТОД ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ **.** ИМЯ-МЕТОДА
ИМЯ-МЕТОДА ::= ИДЕНТИФИКАТОР

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа “класс”.

УНАРНОЕ-ВЫРАЖЕНИЕ ::= УНАРНАЯ-ОПЕРАЦИЯ ВТОРИЧНОЕ-ВЫРАЖЕНИЕ
УНАРНАЯ-ОПЕРАЦИЯ ::= **+** | **-** | **!** | **~**
БИНАРНОЕ-ВЫРАЖЕНИЕ ::= ВЫРАЖЕНИЕ БИНАРНАЯ-ОПЕРАЦИЯ ВЫРАЖЕНИЕ
БИНАРНАЯ-ОПЕРАЦИЯ ::=
***** | **/** | **%** | **+** | **-** | **<<** | **>>** | **in** | **<** | **>** | **<=** | **>=** | **=** | **!=** |
& | **^** | **or** | **xor** | **=>** | **<=>**

Семантика операций и их приоритеты определены ниже в таблице. Типы аргументов каждой операции должны соответствовать операции. Операция “+” для объединения массивов описана в разд. 8.4.

```
МОДИФИКАЦИЯ ::= МОДИФИКАЦИЯ-СТРУКТУРЫ | МОДИФИКАЦИЯ-МАССИВА
МОДИФИКАЦИЯ-СТРУКТУРЫ ::= ВЫРАЖЕНИЕ with АГРЕГАТ
МОДИФИКАЦИЯ-МАССИВА ::=
        ВЫРАЖЕНИЕ with АГРЕГАТ |
        ВЫРАЖЕНИЕ with ОПРЕДЕЛЕНИЕ-МАССИВА-ПО-ЧАСТЯМ |
        ВЫРАЖЕНИЕ with (ИЗОБРАЖЕНИЕ-ТИПА-МНОЖЕСТВА : ВЫРАЖЕНИЕ)
```

В структурном значении выражения, обозначающего массив или структуру, меняются компоненты, представленные АГРЕГАТОМ, ОПРЕДЕЛЕНИЕМ-МАССИВА-ПО-ЧАСТЯМ (см. разд. 8.3) или ИЗОБРАЖЕНИЕМ-ТИПА-МНОЖЕСТВА (см. разд. 7.4). Значением операции является обновленный массив или структура.

```
type ar0_4 = array (int, 0..4);
ar0_4 ones = [ 1, 1, 1, 2, 1 ];
ar0_4 ones1 = ones [3: 1]; // [ 1, 1, 1, 1, 1 ]
type Vec(nat n) = array (real, 1..n);
perm(nat n, Vec(n) b, nat m, k : Vec(n) b')
{ b' = b with [k: b [m], m: b [k]] };
```

Примеры модификации массивов

```
УСЛОВНОЕ-ВЫРАЖЕНИЕ ::= ВЫРАЖЕНИЕ ? ВЫРАЖЕНИЕ : ВЫРАЖЕНИЕ
```

Первое выражение имеет тип "логический". Если первое ВЫРАЖЕНИЕ завершается идентификатором, то идентификатор и последующий символ **"?"** должны быть разделены пробелом.

```
ВЫРАЖЕНИЕ ::= ВТОРИЧНОЕ-ВЫРАЖЕНИЕ |
              УНАРНОЕ-ВЫРАЖЕНИЕ |
              БИНАРНОЕ-ВЫРАЖЕНИЕ |
              УСЛОВНОЕ-ВЫРАЖЕНИЕ
```

^	Возведение в степень
+, -, !, ~	Унарные плюс и минус, логическое отрицание и поэлементное дополнение для множеств, побитовое дополнение для ограниченных целых
***, /, %**	Арифметическое умножение, деление и остаток от целочисленного деления
+, -	Арифметическое сложение и вычитание; объединение и вычитание множеств; конкатенация списков и строк; объединение массивов с непересекающимися типами индексов
<<, >>	Побитовый (поэлементный) сдвиг влево и вправо для целых
In	Проверка вхождения элемента в множество
<, >, <=, >=	Операции арифметического сравнения
=, !=	Проверка на равенство и неравенство
&	Пересечение множеств, логическая конъюнкция, побитовое И для ограниченных целых
Xor	Симметрическая разность для множеств, исключительное ИЛИ для логических выражений и ограниченных целых (побитовое)
Or	~~Объединение множеств~~, логическая дизъюнкция, побитовое ИЛИ для ограниченных целых
=>	Импликация
<=>	Логическое тождество
? :	Трёхместная условная операция

Таблица 2 Операции в порядке уменьшения приоритета

7. Типы

Всякая переменная имеет некоторый тип. Тип может быть атрибутом языковой конструкции. Семантика конструкции налагает определенные ограничения на значения типов. Для каждой подконструкции некоторой конструкции определяется *позиция* ее вхождения внутри конструкции. Это, например, позиции операндов операций, позиции правой и левой частей в операторе присваивания, позиция фактического параметра в вызове предиката и др. При наличии в конструкции двух позиций типы этих позиций должны быть *согласованы*. Частным случаем согласованности является *совместимость* типов. Например, тип выражения правой части оператора присваивания должен быть совместим с типом переменной левой части оператора присваивания, но не наоборот. Реализуется неявное преобразование (называемое *приведением*) значения совместимого типа к типу, требуемому позицией.

Тип T является *вложенным* в тип S, если любое значение типа T являются также значением типа S, т.е. $T \subseteq S$. Совместимость и согласованность типов определяются следующими правилами:

- Тип T_1 будем называть *совместимым* с типом T_2, если $T_1 \subseteq T_2$.
- Тип T_1 *тождественен* типу T_2, если $T_1 \subseteq T_2$ и $T_2 \subseteq T_1$.
- Типы T_1 и T_2 *согласованы*, если они имеют общую мажоранту, т.е. тип $\exists$ T. $T_1 \subseteq T$ & $T_2 \subseteq T$.

```
ОПИСАНИЕ-ТИПА   ::=
          type ИМЯ-ТИПА [( ПАРАМЕТРЫ-ТИПА )] = ИЗОБРАЖЕНИЕ-ТИПА |
          ПРЕДОПИСАНИЕ-ТИПА
ИМЯ-ТИПА        ::= ИДЕНТИФИКАТОР
ПАРАМЕТРЫ-ТИПА  ::=
          ИЗОБРАЖЕНИЕ-ТИПА [:blank:] ИДЕНТИФИКАТОР (, ИДЕНТИФИКАТОР)*
              [, ПАРАМЕТРЫ-ТИПА]
```

Описание типа связывает имя типа с его изображением. Тип может быть параметризован, т.е. зависеть от набора значений переменных и типов, указанных в качестве параметров.

```
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.

```
ИЗОБРАЖЕНИЕ-ТИПА ::=
      ИЗОБРАЖЕНИЕ-ПРИМИТИВНОГО-ТИПА |
      ИЗОБРАЖЕНИЕ-ТИПА-ПО-ИМЕНИ |
      ИЗОБРАЖЕНИЕ-ТИПА-ПРЕДИКАТА |
      ИЗОБРАЖЕНИЕ-ПОДТИПА |
      ИЗОБРАЖЕНИЕ-ПЕРЕЧИСЛЕНИЯ |
      ИЗОБРАЖЕНИЕ-СТРУКТУРНОГО-ТИПА |
      ИЗОБРАЖЕНИЕ-КЛАССА |
      ИЗОБРАЖЕНИЕ-ГРАФОВЫХ ТИПОВ
ИЗОБРАЖЕНИЕ-ПРИМИТИВНОГО-ТИПА ::=
      nat [(РАЗРЯДНОСТЬ-ЦЕЛЫХ)]
    | int [(РАЗРЯДНОСТЬ-ЦЕЛЫХ)]
    | real [(РАЗРЯДНОСТЬ-ВЕЩЕСТВЕННЫХ)]
    | bool
    | char
```

Тип **nat** является подмножеством неотрицательных чисел типа **int**, **char** – множество символов некоторого алфавита. Конкретные представления примитивных типов даются в императивном расширении языка (см. разд. 12).

Разрядность определяет максимальное число битов в представлении значений типа. Указание разрядности в изображении примитивного типа является частью императивного расширения языка.

Значения типа **nat** совместимы с типом **int**, а **int** - с типом **real**. Значения типа меньшей разрядности совместимы с аналогичными типами большей разрядности.

ИЗОБРАЖЕНИЕ-ТИПА-ПО-ИМЕНИ ::=
[ИМЯ-МОДУЛЯ **.**] ИМЯ-ТИПА [**(** АРГУМЕНТЫ **)**]

Ранее определенный тип может быть представлен своим именем. Изображение параметризованного типа представляется с конкретными параметрами. АРГУМЕНТЫ, представленные списком переменных, могут быть опущены, если все эти переменные перечислены в ОПИСАНИИ-КОНТЕКСТА (разд. 4).

ИЗОБРАЖЕНИЕ-ТИПА-КАК-ПАРАМЕТРА ::= **type** ИМЯ-ТИПА

Объявление типа в качестве параметра описания типа или определения предиката реализуется описателем **type**. В качестве обозначения типа может использоваться выражение типа **type**, которым, например, может являться параметр предиката или параметризованного типа. Например:

```
type Point (type T) = struct (T x, y);
type Polyline (type T) = list (Point (T));

decimate (type T, list (T) data : list (T) data') {
  data' =  (data = nil) ?
           nil :
           data.car + (len(data) < 3 ? nil : decimate(T, data.cdr.cdr))
};

reverse (type T, list (T) data : list (T) data') {
  data' =  (data = nil) ?  nil :  reverse (T, data.cdr) + data.car
};

Polyline (real) line = [[ (0.0, 0.0), (0.1, 0.2), (0.2, 0.5),
                          (0.3, 0.6), (0.4, 0.3), (0.5, 0.0) ]];

Polyline (real) line1 = reverse (decimate (real, line));
                    // [[ (0.4, 0.3), (0.2, 0.5), (0.0, 0.0) ]]
```

Пример использования типов в качестве параметров

В примере выше идентификатор T использован в качестве типа-параметра, а тип real – как фактический параметр.

```
squareVec (type T, list (T) data : list (T) data') {
  data' =  (data = nil) ?  nil :  data.car ^ 2 + squareVec (T, data.cdr)
};

list (int) squares = squareVec (int, [[ 0, 1, 2, 3, 4 ]] ); // [[ 0, 1, 4, 9, 16 ]]
```

Выше приведен еще один пример использования типа T в качестве параметра предиката.

ИЗОБРАЖЕНИЕ-ТИПА-ПРЕДИКАТА ::=
predicate ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА

При описании параметров предиката достаточно указать только типы параметров. Имена параметров могут быть опущены. Изображение типа предиката используется для описания переменной предикатного типа.

```
type int_list  = list (int);
map_list (int_list src, predicate (int : int) func : int_list dest) {
   dest =  (src = nil) ?  nil :  func (src.car) + map_list (src.cdr, func))
};
int_list numbers = [[ 0, 1, 2, 3, 4, 5 ]];
int_list squares = map_list (numbers, predicate (int a : int b) { b = a * a; });
// squares = [[ 0, 1, 4, 9, 16, 25 ]]
```

Пример использования предикатного типа

```
ИЗОБРАЖЕНИЕ-ПОДТИПА ::=
        subtype ( ОПИСАНИЕ-ПЕРЕМЕННОЙ : ФОРМУЛА ) |
        ИЗОБРАЖЕНИЕ-ДИАПАЗОНА
```

Конструкция subtype(T x: формула) определяет подмножество типа T. Описание переменной x действует в пределах этой конструкции. Множество значений переменной x, для которых ФОРМУЛА истина, является определяемым подтипом типа T. Если ФОРМУЛА содержит другие переменные, то все они являются параметрами изображаемого типа. Конструкция ОПИСАНИЕ-ПЕРЕМЕННОЙ определена в разд. 3.3.

```
ИЗОБРАЖЕНИЕ-ДИАПАЗОНА  ::= ВЫРАЖЕНИЕ .. ВЫРАЖЕНИЕ
```

Изображение диапазона является подмножеством типа **int**, **enum**, или **char**. Изображение типа subtype(int i: i>=1 & i<=n+1) эквивалентно изображению диапазона: 1..n+1

```
type Odd = subtype (int n: n % 2 = 1);
type diap10_20 = 10..20;
```

```
ИЗОБРАЖЕНИЕ-СТРУКТУРНОГО-ТИПА ::=
    ИЗОБРАЖЕНИЕ-ТИПА-СТРУКТУРЫ |
    ИЗОБРАЖЕНИЕ-ТИПА-ОБЪЕДИНЕНИЯ |
    ИЗОБРАЖЕНИЕ-ТИПА-СПИСКА |
    ИЗОБРАЖЕНИЕ-ТИПА-СТРОКИ |
    ИЗОБРАЖЕНИЕ-ТИПА-МНОЖЕСТВА  |
    ИЗОБРАЖЕНИЕ-ТИПА-МАССИВА
ИЗОБРАЖЕНИЕ-ТИПА-СТРУКТУРЫ ::= struct ( ОПИСАНИЕ-ПОЛЕЙ )
ОПИСАНИЕ-ПОЛЕЙ  ::=
    ИЗОБРАЖЕНИЕ-ТИПА СПИСОК-ИМЕН-ПОЛЕЙ  [, ОПИСАНИЕ-ПОЛЕЙ]
СПИСОК-ИМЕН-ПОЛЕЙ   ::=  ИМЯ-ПОЛЯ [, СПИСОК-ИМЕН-ПОЛЕЙ]
```

Значение типа структуры состоит из совокупности значений полей структуры. Имена полей должны быть различны. Число полей должно быть не менее двух.

```
type Point = struct ( int x, y );
Point pt = (10, 20);
```

Пример определения структуры

7.1. Тип объединения

```
ИЗОБРАЖЕНИЕ-ТИПА-ОБЪЕДИНЕНИЯ  ::= union ( ОПИСАНИЯ-КОНСТРУКТОРОВ )
ОПИСАНИЯ-КОНСТРУКТОРОВ ::=
     ОПИСАНИЕ-КОНСТРУКТОРА (, ОПИСАНИЕ-КОНСТРУКТОРА)+
```

Значением типа объединения является значение одного из конструкторов, перечисленных в списке описаний конструкторов. Число конструкторов должно быть не меньше двух.

```
ОПИСАНИЕ-КОНСТРУКТОРА ::= ИМЯ-КОНСТРУКТОРА [ ( ОПИСАНИЕ-ПОЛЕЙ ) ]
ИМЯ-КОНСТРУКТОРА ::= ИДЕНТИФИКАТОР
```

Поля, определяемые в круглых скобках, являются полями объединения. Значения этих полей используются в качестве аргументов конструктора. Имена полей объединения по всем конструкторам должны быть различны.

Работа с объектами типа объединения реализуется с помощью следующих конструкций: конструктор, распознаватель и поле объединения. Все они являются ПЕРВИЧНЫМИ-ВЫРАЖЕНИЯМИ (см. разд. 6).

```
КОНСТРУКТОР ::=  ИМЯ-КОНСТРУКТОРА [ ( СПИСОК-ВЫРАЖЕНИЙ ) ]
```

Выражения, подставляемые в качестве аргументов КОНСТРУКТОРА, по их числу и типам должны соответствовать ОПИСАНИЯМ-ПОЛЕЙ в соответствующем ОПИСАНИИ-КОНСТРУКТОРА. Правила соответствия такие же, как для вызова предиката. Значением конструкции КОНСТРУКТОР является имя конструктора вместе со значениями его аргументов, если аргументы присутствуют.

```
РАСПОЗНАВАТЕЛЬ ::=  ИМЯ-РАСПОЗНАВАТЕЛЯ ( ВЫРАЖЕНИЕ )
ИМЯ-РАСПОЗНАВАТЕЛЯ ::=  ИМЯ-КОНСТРУКТОРА ?
```

Символ “**?**” является частью имени распознавателя и пишется слитно с именем конструктора, без пробела. ВЫРАЖЕНИЕ имеет тип объединения. Распознаватель является функцией типа **bool**. Значение РАСПОЗНАВАТЕЛЯ истинно, если значением ВЫРАЖЕНИЯ является конструктор с именем ИМЯ-КОНСТРУКТОРА.

```
ПОЛЕ-ОБЪЕДИНЕНИЯ   ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ . ИМЯ-ПОЛЯ
```

Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является переменная типа “объединение”. Значением ПОЛЯ-ОБЪЕДИНЕНИЯ является значение поля с именем ИМЯ-ПОЛЯ в структуре переменной типа “объединение”. Использование конструкции ПОЛЕ-ОБЪЕДИНЕНИЯ корректно лишь в случае истинности распознавателя ИМЯ-КОНСТРУКТОРА**?(** ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ **)** для конструктора, к которому относится поле с именем ИМЯ-ПОЛЯ, в противном случае исполнение конструкции не определено.

Имена конструкторов и распознавателей являются глобально определенными в теле модуля, содержащего ИЗОБРАЖЕНИЕ-ТИПА-ОБЪЕДИНЕНИЯ.

```
ОПРЕДЕЛЕНИЕ-КОНСТРУКТОРА ::=
        ИМЯ-КОНСТРУКТОРА [ ( ОПРЕДЕЛЕНИЕ-ПОЛЕЙ-КОНСТРУКТОРА ) ]
ОПРЕДЕЛЕНИЕ-ПОЛЕЙ-КОНСТРУКТОРА ::=
        ОПРЕДЕЛЕНИЕ-ПОЛЯ-КОНСТРУКТОРА
                [, ОПРЕДЕЛЕНИЕ-ПОЛЕЙ-КОНСТРУКТОРА]
ОПРЕДЕЛЕНИЕ-ПОЛЯ-КОНСТРУКТОРА ::=
        [ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ] ИДЕНТИФИКАТОР
```

Конструкция ОПРЕДЕЛЕНИЕ-КОНСТРУКТОРА используется в качестве АЛЬТЕРНАТИВЫ после **case** в операторе выбора (см. разд. 5) для выражения типа объединения. Значением АЛЬТЕРНАТИВЫ является конструктор с именем ИМЯ-КОНСТРУКТОРА и набором переменных, обозначающих поля данного конструктора. Эти переменные являются локальными в ОПЕРАТОРЕ-АЛЬТЕРНАТИВЫ, исполняемом для

данной АЛЬТЕРНАТИВЫ. Типы переменных могут не указываться, поскольку они определяются из ИЗОБРАЖЕНИЯ-ТИПА-ОБЪЕДИНЕНИЯ.

```
type int_tree = union (
   leaf (int val),
   node (int_tree lhs, int v, int_tree rhs)
);

int_tree foo;

// ...

switch (foo) {
   case leaf(v0): print("leaf", v0)
   case node(l, v1, r): print("node with ", v1)
};

int_tree one_leaf = leaf(42);
int_tree small_tree = node(leaf(1), 2, node(leaf(3), 4, leaf(5)));
// small_tree задаёт следующее дерево:
//    2
//   / \
//  1   4
//     / \
//    3   5
```

Пример использования объединений

```
ИЗОБРАЖЕНИЕ-ПЕРЕЧИСЛЕНИЯ ::=
      enum ( ИДЕНТИФИКАТОР (, ИДЕНТИФИКАТОР)* )
```

Тип перечисления является частным случаем типа объединения с конструкторами без аргументов. Описатель **enum** трактуется как эквивалент **union**.

```
type fruit = enum (apple, orange, banana);
```

Пример перечисления

7.2. Тип списка

Структура вида “список” представляется как упорядоченная совокупность однородных элементов. Тип списка из элементов типа T определяется следующим описанием:

```
type list (type T) = union (
      nil,
      cons (T car, list(T) cdr)
);
```

Определение списка

Тип list считается определенным в языке P и не требует описания в программе. Имена list, nil и cons глобально определены в предикатной программе.

Конструктор nil определяет пустой список, не содержащий элементов. Остальные значения типа list представляются конструктором cons(x, s), где элемент x есть первый элемент списка - голова списка, а s

есть оставшаяся часть списка - хвост списка. Функция len(s) определяет число элементов списка s. Операция конкатенации s1 + s2 определяет список, состоящий из элементов списка s1, за которыми следуют элементы списка s2. Аргументами конкатенации могут быть также элементы списка. Функция last(s) определяет последний элемент непустого списка s; функция prec(s) - оставшуюся часть списка без последнего элемента.

ЭЛЕМЕНТ-СПИСКА ::= ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ **[** ИНДЕКС-ЭЛЕМЕНТА **]**

Данная конструкция определяет значение элемента с указанным индексом в списке. Значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ является список.

ИНДЕКС-ЭЛЕМЕНТА ::= ВЫРАЖЕНИЕ

Циклы **for** и **while** часто используется для представления циклов, реализуемых с помощью операторов перехода, в целях улучшения структуры программы.

Для списка s конструкция s[i] определяет i-ый элемент. Допустимыми являются значения индекса i от 0 до len(s)-1.

ВЫРЕЗКА-СПИСКА ::=
ТЕРМ ТИПА СПИСОК [ИЗОБРАЖЕНИЕ-ДИАПАЗОНА] |
ТЕРМ ТИПА СПИСОК [ВЫРАЖЕНИЕ ..]
ТЕРМ ТИПА СПИСОК ::=ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ

Значением вырезки вида s[m..n] является список, начинающийся элементом с номером m и заканчивающийся элементом с номером n. Элементы списка нумеруются с нуля. Значением вырезки будет пустой список (значение nil), если m больше номера последнего элемента, либо если n < m. Значением вырезки вида s[m..] является список, начинающийся элементом с номером m и включающий все элементы до конца списка.

```
type int_list   = list (int);
int_list s      = [[ 1, 1, 2, 3, 5, 8 ]];
int s_car       = s.car;       // s_car = 1
int_list s_cdr  = s.cdr;       // s_cdr = [ 1, 2, 3, 5, 8 ]
int_list ss     = s + [[ 13, 21 ]];  // ss = [ 1, 1, 2, 3, 5, 8, 13, 21 ]
int count       = len (s);     // count = 6
int e           = s[3];        // e = 3
```

Примеры операций со списком

7.3. Строковый тип

ИЗОБРАЖЕНИЕ-ТИПА-СТРОКИ ::= **string**

Строковый тип string является предопределенным в языке P. Его определение имеет вид:

```
type string = list(char);
```

Набор средств, определенных в разделе 7.2 для списков, применим также для строк с некоторыми ограничениями. В дополнении к этому в языке P определены строковые константы (см. разд. 2).

Основным представлением строкового объекта является массив литер, завершающийся нулем, причем нуль не входит в значение строки. Иначе говоря, для строкового типа фактически действует следующее определение:

type string = **subtype**(list(**char**) s**:** s != nil & last(s) = 0[1]);

Проверка строки s на пустоту реализуется операцией s.car = 0, а не s = nil. В принципе, возможна реализация, в которой конструктор nil кодируется значением из единственного нулевого элемента, однако такое решение приведет к потере эффективности. В итоге, типы list и string несовместимы: со строковым объектом нельзя работать как со списком, в частности, нельзя подставлять строковый объект параметром типа list. Как следствие, библиотеки для списков неприменимы для строковых объектов.

Вводятся дополнительные конструкции для строковых объектов. Для определения числа элементов строки s вместо функции len(s) используется length(s). Конструктор nil, распознаватель nil?(s), а также отношения s = nil и s != nil не используются для строковых объектов. В качестве пустой строки используется конструктор empty, значением которого является строка из единственного нулевого элемента.

7.4. Тип множества

ИЗОБРАЖЕНИЕ-ТИПА-МНОЖЕСТВА ::=
set (ИЗОБРАЖЕНИЕ-БАЗОВОГО-ТИПА **)**
ИЗОБРАЖЕНИЕ-БАЗОВОГО-ТИПА ::= ИЗОБРАЖЕНИЕ-ТИПА

Тип множества есть множество всех подмножеств некоторого конечного базового типа. Имеется унарная операция ~ дополнения множества. Определены: операция "+" объединения множеств, операция "-" разности множеств, а также вычитания элемента из множества, операция & пересечения множеств, операция ~~or объединения множеств~~, операция in принадлежности элемента множеству (см. разд. 6). Число элементов в множестве реализуется функцией len.

Гиперфункция **pick**(s : : x, s1) недетерминировано выбирает произвольный элемент x из множества s, если он существует. Множество s1 – это оставшееся множество без элемента x, то есть $s = s1 \cup \{x\}$. Первая ветвь – это результат выбора гиперфункции, когда множество s пусто. Заметим, что в случае непустого множества s гиперфункция **pick** быстрее, чем последовательное применение двух действий: проверка на пустоту и выбор произвольного элемента из s. Гиперфункция **pick** определяется следующей спецификацией:

```
pick(set(T) s : : T x, set(T) s1)
    pre 1: s=∅
    post 2: s=s1∪{x} & x∈s & x∉s1
```

Предусловие по первой ветви: $s=\varnothing$. Предусловие по второй ветви: $s\neq\varnothing$ (оно не указывается, поскольку является дополнением к предусловию первой ветви). Постусловие по первой ветви отсутствует, так как там нет результатов.

Операция mask преобразует множество в целое значение. Операция bits реализует обратную операцию преобразования целого в множество.

[1] В операции сравнения предполагается неявное приведение last(s) к типу **int**.

```
type int_set_t       = set (1..1000);
int_set_t pow_2      = { 0, 2, 4, 8, 16, 32 };
bool is_pow_2        = 4 in pow_2; // is_pow_2 = true
int_set_t more_pow_2 = pow_2 + { 64, 128 };
```

Пример использования множеств

7.5. Тип класса

```
ИЗОБРАЖЕНИЕ-КЛАССА ::=
        class [extends ИМЯ-СУПЕРКЛАССА]{ ОПИСАНИЯ-КЛАССОВ }
ОПРЕДЕЛЕНИЕ-КЛАССА ::=
        class ИМЯ-КЛАССА [( ПАРАМЕТРЫ-ТИПА )]
                [extends ИМЯ-СУПЕРКЛАССА]{ ОПИСАНИЯ-КЛАССА }
ИМЯ-КЛАССА ::= ИМЯ-ТИПА
ИМЯ-СУПЕРКЛАССА ::= ИМЯ-ТИПА
```

ОПРЕДЕЛЕНИЕ-КЛАССА эквивалентно следующему описанию типа:

```
type ИМЯ-КЛАССА [( ПАРАМЕТРЫ-ТИПА )] = ИЗОБРАЖЕНИЕ-КЛАССА
```

Класс представляет независимую область локализации имен, определяемых ОПИСАНИЯМИ-КЛАССА.

```
ОПИСАНИЯ-КЛАССОВ ::= ОПИСАНИЕ-КЛАССА  [ ; ОПИСАНИЯ-КЛАССОВ ]
ОПИСАНИЕ-КЛАССА ::= ОПИСАНИЕ | ОПИСАНИЕ-КОНСТРУКТОРА-КЛАССА
```

Элементы класса - поля и методы класса. Поля класса представлены переменными, определяемыми описаниями переменных внутри ОПИСАНИЙ-КЛАССА. Методы класса - предикаты, представленные определениями и спецификациями предикатов внутри ОПИСАНИЙ-КЛАССА. Методами класса являются также определенные в классе процессы.

При наличии подконструкции **extends** в описании класса элементы класса определяются как продолжение элементов суперкласса.

Объект класса - значение переменной типа класса. Значением объекта является набор полей класса аналогично значению структурного типа.

Доступ к элементу класса реализуется через объект класса с помощью конструкции ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ**.**имя-элемента, где объект является значением ПЕРВИЧНОГО-ВЫРАЖЕНИЯ. Доступ к элементу класса внутри класса осуществляется непосредственно по имени элемента. Для обозначения объекта, определяемого данным классом, используется имя **this**.

```
ОПИСАНИЕ-КОНСТРУКТОРА-КЛАССА ::=
        ИМЯ-КЛАССА ([ОПИСАНИЯ-АРГУМЕНТОВ]) { ТЕЛО-КОНСТРУКТОРА }
ТЕЛО-КОНСТРУКТОРА  ::= [ ОПЕРАТОР ]
```

Создание нового объекта класса реализуется через вызов конструктора, значением которого является созданный объект. Значения полей класса формируются в теле конструктора, а также начальной инициализацией полей при их описании. Если для вызова ИМЯ-КЛАССА( ) нет соответствующего конструктора с пустым списком аргументов, то такой конструктор считается определенным и его тело пусто. В результате работы конструктора все поля должны быть сформированы. Конструктор класса содержит явный или неявный вызов соответствующего конструктора для суперкласса.

8. Массивы

8.1. Описание типа массива

```
ИЗОБРАЖЕНИЕ-ТИПА-МАССИВА  ::=
        array ( ИЗОБРАЖЕНИЕ-ТИПА-ЭЛЕМЕНТА , ИЗМЕРЕНИЯ-МАССИВА )
ИЗОБРАЖЕНИЕ-ТИПА-ЭЛЕМЕНТА ::= ИЗОБРАЖЕНИЕ-ТИПА
ИЗМЕРЕНИЯ-МАССИВА  ::=
        ИЗОБРАЖЕНИЕ-ТИПА-ИНДЕКСА [, ИЗМЕРЕНИЯ-МАССИВА]
ИЗОБРАЖЕНИЕ-ТИПА-ИНДЕКСА  ::= ИЗОБРАЖЕНИЕ-ТИПА
```

Значение массива состоит из совокупности элементов. Каждый элемент доступен по набору индексов в массиве: для массива A элемент с набором индексов m есть A[m]. Число индексов совпадает с числом измерений массива. Тип каждого индекса определяет совокупность допустимых значений индекса и должен быть конечным.

Имеется операция "+" для объединения двух массивов-операндов в один массив при условии, что типы элементов совпадают, а множества наборов индексов операндов совместимы по типам и не пересекаются.

```
type int_vec = array (int, 1..5);
type int_mtx = array (int, 1..2, 1..3);

int_vec v = [ 3, 1, 4, 1, 5 ];
int_vec w = for (x) x*x;
int v_2 = v [2]; // v_2 = 1

int_mtx m = [ [ 1, 2, 3 ], [ 104, 5, 6 ] ];
int e = m [2, 3]; // e = 6
```

Пример описаний массивов и их инициализации

8.2. Вырезка массива

```
ВЫРЕЗКА-МАССИВА   ::=
        ВЫРАЖЕНИЕ-МАССИВ [ СУЖЕННЫЙ-НАБОР-ТИПОВ-ИНДЕКСОВ ]
ВЫРАЖЕНИЕ-МАССИВ  ::= ВЫРАЖЕНИЕ
СУЖЕННЫЙ-НАБОР-ТИПОВ-ИНДЕКСОВ ::=  НАБОР-ИНДЕКСОВ
НАБОР-ИНДЕКСОВ ::= ЭЛЕМЕНТ-НАБОРА-ИНДЕКСОВ [, НАБОР-ИНДЕКСОВ]
ЭЛЕМЕНТ-НАБОРА-ИНДЕКСОВ ::=
        ЗНАЧЕНИЕ-ИНДЕКСА | ИЗОБРАЖЕНИЕ-ДИАПАЗОНА
ЗНАЧЕНИЕ-ИНДЕКСА ::= ВЫРАЖЕНИЕ
```

Вырезка массива определяет массив как часть некоторого массива заданием подмножества наборов индексов. Достаточно задать сужение хотя бы по одному измерению массива. В частности, допустимо сужение к единственному значению индекса. Однако вырезка массива не может превратиться в единственный элемент массива. Результатом исполнения вырезки массива является переменная типа “массив”. Нового массива при этом не создается.

8.3. Определение массива

```
ОПРЕДЕЛЕНИЕ-МАССИВА ::=
        ОПРЕДЕЛЕНИЕ-ИНДЕКСОВ   ОПРЕДЕЛЕНИЕ-ЭЛЕМЕНТА-МАССИВА |
        АГРЕГАТ-МАССИВ |
        ОПРЕДЕЛЕНИЕ-МАССИВА-ПО-ЧАСТЯМ
ОПРЕДЕЛЕНИЕ-МАССИВА-ПО-ЧАСТЯМ ::=
        ОПРЕДЕЛЕНИЕ-ИНДЕКСОВ   ОПРЕДЕЛЕНИЕ-ЧАСТЕЙ-МАССИВА
```

Результатом вычисления ОПРЕДЕЛЕНИЯ-МАССИВА является значение массива. Вычисление реализуется итерацией по всевозможным значениям набора индексов массива. Для каждого набор индексов вычисляется соответствующий элемент массива. Вычисление значений разных элементов может проводиться параллельно.

```
ОПРЕДЕЛЕНИЕ-ЭЛЕМЕНТА-МАССИВА ::=  ВЫРАЖЕНИЕ
```

ВЫРАЖЕНИЕ для элемента массива зависит от индексов, указанных в ОПРЕДЕЛЕНИИ-ИНДЕКСОВ.

```
ОПРЕДЕЛЕНИЕ-ИНДЕКСОВ ::= for ( ЗАДАНИЕ-ИНДЕКСОВ )
ЗАДАНИЕ-ИНДЕКСОВ  ::= ОПРЕДЕЛЕНИЕ-ИНДЕКСА [ , ЗАДАНИЕ-ИНДЕКСОВ]
ОПРЕДЕЛЕНИЕ-ИНДЕКСА  ::= [ИЗОБРАЖЕНИЕ-ТИПА-ПЕРЕМЕННОЙ] ИНДЕКС
ИНДЕКС ::= ИДЕНТИФИКАТОР
```

Переменные, обозначающие индексы, локальны в ОПРЕДЕЛЕНИИ-МАССИВА. При определении индекса обычно используется описатель **var** (см. разд. 4). Указание типа индекса требуется в редких случаях, когда тип массива (в том числе и типы индексов массива) трудно определить из позиции, в которой находится ОПРЕДЕЛЕНИЕ-МАССИВА. Отметим, что в ОПРЕДЕЛЕНИИ-ИНДЕКСА тип индекса, если он задан явно, должен точно покрывать множество значений индекса; обычно, это диапазон типа **int**.

```
type ar1_5 = array (int, 1..5);
ar1_5 squ;
squ = for (var i) i*i;
ar1_5 r = for (var i) 100 - i;
```

Примеры определения массивов

```
ОПРЕДЕЛЕНИЕ-ЧАСТЕЙ-МАССИВА ::=
    { (ОПРЕДЕЛЕНИЕ-ЧАСТИ-МАССИВА)+
          [default : ОПРЕДЕЛЕНИЕ-ЭЛЕМЕНТА-МАССИВА ] }
ОПРЕДЕЛЕНИЕ-ЧАСТИ-МАССИВА  ::=
    case ИНДЕКСЫ-ЧАСТИ : ОПРЕДЕЛЕНИЕ-ЭЛЕМЕНТА-МАССИВА
ИНДЕКСЫ-ЧАСТИ   ::=
    ЭЛЕМЕНТ-НАБОРА-ИНДЕКСОВ |
    ( НАБОР-ИНДЕКСОВ ) |
    ( ЭЛЕМЕНТ-НАБОРА-ИНДЕКСОВ [, ИНДЕКСЫ-ЧАСТИ] ) |
    ( ( НАБОР-ИНДЕКСОВ ) [, ИНДЕКСЫ-ЧАСТИ] )
```

ИНДЕКСЫ-ЧАСТИ определяют некоторое подмножество на произведении типов индексов. Эти подмножества не должны пересекаться для разных ОПРЕДЕЛЕНИЙ-ЧАСТЕЙ-МАССИВА. Определение элементов массива для наборов индексов, не принадлежащих ни одной из указанных частей массива, реализуется частью **default**. При отсутствии части **default** объединение подмножеств наборов индексов для разных частей должно совпадать с полным множеством наборов индексов. Вычисление элементов массива по каждой из частей массива проводится независимо, возможно параллельно.

Если ОПРЕДЕЛЕНИЕ-ЧАСТЕЙ-МАССИВА используется в операции модификации массива (см. разд. 6), то часть **default** отсутствует.

```
type Ar(nat k) = array (real, 1..k);
F (nat n, Ar(n) x: Ar(n+1) x')
{ x' = for (var j) { case 1..n : x[j] + 1 case n + 1 : 0 } }
```

Пример определения массива по частям

```
type MATR(nat k) = array(real, 1..k, 1..k);
perm_lines(nat n, MATR(n) a, nat k, m : MATR(n) a' )
pre  1 <= k < m <= n
{   a' = a  with for (var i, j) {
        case (k, 1..n): a[m, j]
        case (m, 1..n): a[k, j]
    }
}
```

Пример перестановки двух строк матрицы

В матрице **a** переставляются местами строки с номерами **k** и **m**. Перестановка реализуется применением операции модификации (см. разд. 6) двух строк в массиве **a**; остальные строки остаются

неизменными. Задание нового значения двух строк матрицы реализуется конструкцией ОПРЕДЕЛЕНИЕ-ЧАСТЕЙ-МАССИВА.

8.4. Объединение массивов

Операндами операции "**+**" объединения массивов:

ВЫРАЖЕНИЕ-МАССИВ **+** ВЫРАЖЕНИЕ-МАССИВ

являются выражения, значениями которых являются массивы. Объединяемые массивы должны иметь совпадающие типы элементов и непересекающиеся типы индексов, причем типы индексов должны принадлежать некоторому общему типу. Результатом операции является массив. Тип индексов результирующего массива есть объединение типов индексов операндов, а тип элементов тот же, что и у операндов. Произвольный элемент результирующего массива есть либо элемент первого массива-операнда, либо элемент второго.

9. Графы [13]

Граф определяется множеством *вершин* и множеством *ребер*. Ребро – это пара вершин графа. Вершины x и y ребра (x, y) называются смежными (соседними). Ребро соединяет x и y и инцидентно вершинам x и y. Вершины x и y также называются концевыми ребра (x, y). Степень вершины – это количество инцидентных ей рёбер.

```
ИЗОБРАЖЕНИЕ-ГРАФОВЫХ ТИПОВ
ИЗОБРАЖЕНИЕ-ТИПОВ-ВЕРШИН И РЕБЕР ГРАФА ::= ИЗОБРАЖЕНИЕ-ТИПА-ВЕРШИН |
                                            ИЗОБРАЖЕНИЕ-ТИПА-РЕБЕР
ИЗОБРАЖЕНИЕ-ТИПА-ВЕРШИН ::= node
ИЗОБРАЖЕНИЕ-ТИПА-РЕБЕР ::= struct(source: node, target: node)
```

В языке P любая вершина имеет тип **node** и по умолчанию совпадает с типом **nat**. Если для конкретного приложения требуется другой базовый тип для набора вершин, отличный от **nat**, пользователю разрешается изменить тип вершины. Например, пользователь может ввести следующее описание:

```
type node = subtype(int i: i>=1 & i<=10);
```

Для множества вершин и множества ребер используются стандартные типы nodeset и edgeset.

```
ИЗОБРАЖЕНИЕ-МНОЖЕСТВА-ВЕРШИН ::= nodeset
ИЗОБРАЖЕНИЕ-МНОЖЕСТВА-РЕБЕР ::= edgeset
```

Определения типов следующие:

```
type nodeset = set(node);
type edgeset = set(struct(source: node, target: node));
```

Для произвольного графа, определенного множеством вершин **nodes** и множеством ребер **edges**, существует дополнительное условие, определяемое предикатом **is_graph**: концевые вершины каждого ребра принадлежат множеству **nodes**. Кроме того, множества **nodes** и **edges** должны быть конечными:

```
formula is_graph(nodeset nodes, edgeset edges) =
                                    finite(nodes) & finite(edges) &
                                        ∀ node a, b. (a, b) ∈ edges => a ∈ nodes & b ∈ nodes;
```

Эффективность программ обработки графов достигается путем применения оптимизирующих трансформаций операций с множествами, учитывающих особенности конкретного приложения.

```
formula unordered (nodeset nodes, edgeset edges) =
    is_graph(nodes, edges) & ∀ node a, b. (a, b)∈edges => a ≠ b & (b, a)∉edges;
nodeset V;
edgeset E;
Degree( : array(nat, V) d)
    pre unordered(V, E)
    post postDeg (V, E, d)
{ NodesDegree( E, for (V k) 0 ) }

NodesDegree(edgeset eds, array(nat, V) deg: array(nat, V) d)
    pre unordered(V, E) & ∀ v ∈ V. deg[v] = degree(v, V, E \ eds)
    post postDeg (V, E, d)
{ pick(eds){  case : deg
              case (node i, node j), edgeset eds1:
                   NodesDegree(eds1, deg with (i: deg[i]+1, j: deg[j]+1) )
}}
```

Пример использования графов (Программа поиска степеней вершин в неориентированном графе)

10. Формулы

Формулы, используемые в качестве предусловий и постусловий предикатов, есть формулы типизированного исчисления предикатов высших порядков. Подформула, входящая в предусловие или постусловие, может быть определена отдельно в виде описания формулы.

```
ОПИСАНИЕ-ФОРМУЛЫ ::=
        formula ИМЯ-ФОРМУЛЫ [ОПИСАНИЯ-ВНЕШНИХ-АРГУМЕНТОВ]
        ( ОПИСАНИЯ-АРГУМЕНТОВ [: ТИП-РЕЗУЛЬТАТА] ) = ФОРМУЛА
        [measure ВЫРАЖЕНИЕ]
ИМЯ-ФОРМУЛЫ    ::= ИДЕНТИФИКАТОР
ТИП-РЕЗУЛЬТАТА  ::= ИЗОБРАЖЕНИЕ-ТИПА
```

ОПИСАНИЕ-ФОРМУЛЫ вводит имя формулы для обозначения конструкции, представленной ФОРМУЛОЙ. Переменные, входящие в ФОРМУЛУ, должны быть указаны в ОПИСАНИЯХ-АРГУМЕНТОВ; см. разд. 3.1. Описание формулы помещается среди описаний модуля программы; см. разд. 4. ФОРМУЛА является выражением типа ТИП-РЕЗУЛЬТАТА. Если ТИП-РЕЗУЛЬТАТА не указан, то ФОРМУЛА имеет тип **bool.** Мера должна быть задана для рекурсивно определяемых формул в случае, когда реализуется дедуктивная верификация предикатной программы с использованием системы PVS [7].

Использование формулы, определенной ОПИСАНИЕМ-ФОРМУЛЫ, в других формулах реализуется через вызов формулы.

```
ВЫЗОВ-ФОРМУЛЫ ::=
        ИМЯ-ФОРМУЛЫ [ВНЕШНИЕ-АРГУМЕНТЫ] ( СПИСОК-ВЫРАЖЕНИЙ )
```

Описание формулы может быть рекурсивным: ФОРМУЛА в правой части описания может содержать рекурсивный вызов этой формулы. Не допускается взаимной рекурсии в описаниях двух разных формул. Другое требование: рекурсивный вызов должен находиться в *позитивной позиции* ФОРМУЛЫ. Понятия позитивной и негативной позиции определены ниже.

ВНЕШНИЕ-АРГУМЕНТЫ являются начальной частью аргументов формулы. Они могут быть опущены в случае указания их в составе ОПИСАНИЯ-КОНТЕКСТА (см. разд. 4).

```
ФОРМУЛА ::= ВЫРАЖЕНИЕ |
            ( ФОРМУЛА ) |
            ВЫЗОВ-ФОРМУЛЫ |
            ! ФОРМУЛА |
            ФОРМУЛА & ФОРМУЛА |
            ФОРМУЛА or ФОРМУЛА |
            ФОРМУЛА => ФОРМУЛА |
            ФОРМУЛА <=> ФОРМУЛА |
            КВАНТОРНАЯ-ЧАСТЬ  ФОРМУЛА
```

Все альтернативы приведенного правила, кроме первых трех, определяют формулы типа **bool**, т.е. предикаты. Операции "**!**", "**&**" и "**or**" имеют тот же смысл, что и для логических выражений. Операция "**=>**" обозначает импликацию, "**<=>**" - логическое тождество.

ВЫРАЖЕНИЕ, находящееся в позиции ФОРМУЛЫ, не может содержать внутри себя ВЫЗОВА-ФУНКЦИИ. Взамен допускается использовать ВЫЗОВ-ФОРМУЛЫ.

```
КВАНТОРНАЯ-ЧАСТЬ ::=
    КВАНТОР  СПИСОК-ПОДКВАНТОРНЫХ-ПЕРЕМЕННЫХ .
       [КВАНТОРНАЯ-ЧАСТЬ]
КВАНТОР ::= КВАНТОР-ВСЕОБЩНОСТИ | КВАНТОР-СУЩЕСТВОВАНИЯ
КВАНТОР-ВСЕОБЩНОСТИ ::=  forall
КВАНТОР-СУЩЕСТВОВАНИЯ ::=  exists
СПИСОК-ПОДКВАНТОРНЫХ-ПЕРЕМЕННЫХ ::=
     ИЗОБРАЖЕНИЕ-ТИПА  ПОДКВАНТОРНАЯ-ПЕРЕМЕННАЯ
        (, ПОДКВАНТОРНАЯ-ПЕРЕМЕННАЯ)*
        [, СПИСОК-ПОДКВАНТОРНЫХ-ПЕРЕМЕННЫХ ]
ПОДКВАНТОРНАЯ-ПЕРЕМЕННАЯ   ::= ИДЕНТИФИКАТОР
```

Приоритеты логических связок в порядке убывания следующие: **!**, **&**, **=>**,(or?) **<=>**, кванторы **forall** и **exists**.

Определим понятие позиции, позитивной или негативной, для произвольного вхождения подформулы. Позиция ФОРМУЛЫ в качестве правой части ОПИСАНИЯ-ФОРМУЛЫ является позитивной. Допустим, формула X есть одна из следующих формул: A & B, A **or** B, !C, C => A, **forall** y. A, **exists** y. A. Для перечисленных случаев подформулы A и B имеют туже позицию, что формула X, а подформула C меняет позицию на противоположную. Все остальные позиции подформул считаются неизвестными.

```
ОПИСАНИЕ-ТЕОРИИ  ::=
        theory ИМЯ-ТЕОРИИ [ ( ОПИСАНИЯ-АРГУМЕНТОВ ) ] = { ТЕЛО-ТЕОРИИ }
ТЕЛО-ТЕОРИИ   ::=  УТВЕРЖДЕНИЕ (;УТВЕРЖДЕНИЕ)*
ИМЯ- ТЕОРИИ ::= ИДЕНТИФИКАТОР
УТВЕРЖДЕНИЕ    ::=
       [ИМЯ-УТВЕЖДЕНИЯ :] ОПИСАТЕЛЬ-УТВЕРЖДЕНИЯ   ФОРМУЛА
ИМЯ-УТВЕЖДЕНИЯ ::= ИДЕНТИФИКАТОР
ОПИСАТЕЛЬ-УТВЕРЖДЕНИЯ  ::=  axiom | lemma |theorem
```

ОПИСАНИЕ-ТЕОРИИ определяет набор лемм (теорем), используемых при доказательстве формул корректности программы применением SMT-решателя. ИМЯ-УТВЕЖДЕНИЯ используется в системе автоматического доказательства для идентификации утверждения. Утверждения с описателем **axiom** считаются априори истинными. Утверждения с описателем **lemma** или **theorem** должны быть доказаны в системе автоматического доказательства.

11. Процессы

Предикатные программы соответствуют классу программ-функций [8] для задач дискретной и вычислительной математики. Программы-процессы возникают во многих приложениях, в частности, при создании систем управления, где программа представлена в виде набора секций [12] и параллельных взаимодействующих процессов. Примерами таких программ являются протоколы и телекоммуникационные системы. В целях спецификации и реализации данного класса программ язык P расширен средствами для описания процессов и секций, передачи сообщений и порождения процессов, работающих параллельно с процессом-родителем.

```
ОПРЕДЕЛЕНИЕ-СЕКЦИИ ::= section ИМЯ-СЕКЦИИ extends ИМЯ-СЕКЦИИ
{ ОПИСАНИЕ-СЕКЦИИ (ОПИСАНИЕ-СЕКЦИИ)* }
ИМЯ-СЕКЦИИ ::= ИДЕНТИФИКАТОР
ОПИСАНИЕ-СЕКЦИИ ::= ОПИСАНИЕ-ТИПА |
                     КОНСТАНТА |
                     ОПИСАНИЕ-ПЕРЕМЕННЫХ |
                     inv ИНВАРИАНТ-СЕКЦИИ |
                     АКСИОМА
АКСИОМА ::= axiom ФОРМУЛА
ИНВАРИАНТ-СЕКЦИИ ::= ФОРМУЛА
```

В секции описываются переменные состояния программы, а также константы и типы. Состояние автоматной программы определяется значениями набора переменных, модифицируемых СЕГМЕНТАМИ АВТОМАТНОЙ-ПРОГРАММЫ. Секция может быть построена расширением другой секции при наличии extends. В секции также помещаются общие инварианты, которые должны быть истинными для всей программы.

```
ОПРЕДЕЛЕНИЕ-ПРОЦЕССА ::=
    process ИМЯ-ПРОЦЕССА [( [ОПИСАНИЯ-АРГУМЕНТОВ] :
                              ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ГИПЕРФУНКЦИИ )]
      { СОСТОЯНИЕ-ПРОЦЕССА  ТЕЛО-ПРОЦЕССА }
ИМЯ-ПРОЦЕССА  ::=  ИДЕНТИФИКАТОР
```

Процесс может быть бесконечным. Если процесс может завершиться, то завершение реализуется по одному или нескольким разным выходам процесса. Результаты возможно завершающегося процесса определяются так же, как у гиперфункции.

```
СОСТОЯНИЕ-ПРОЦЕССА  ::= [ОПИСАНИЕ-ПЕРЕМЕННЫХ ;]
```

Состояние процесса определяется набором переменных, модифицируемых при исполнении процесса.

```
ТЕЛО-ПРОЦЕССА  ::= АВТОМАТНАЯ-ПРОГРАММА |
                   ПАРАЛЛЕЛЬНАЯ-КОМПОЗИЦИЯ
ПАРАЛЛЕЛЬНАЯ-КОМПОЗИЦИЯ  ::=
                   ВЫЗОВ-ПРОЦЕССА [ || ПАРАЛЛЕЛЬНАЯ-КОМПОЗИЦИЯ ]
```

В общем случае процесс определен в виде параллельной композиции последовательных процессов, определяемых в виде АВТОМАТНОЙ-ПРОГРАММЫ. Переменная из состояния процесса является разделяемой, если она доступна в различных процессах параллельной композиции.

```
АВТОМАТНАЯ-ПРОГРАММА  ::= СЕГМЕНТ [ АВТОМАТНАЯ-ПРОГРАММА ]
```

Автоматная программа является автоматом (машиной конечных состояний) и представлена набором сегментов кода. Вершина автомата - начало сегмента, гипердуга автомата - сегмент. Исполнение сегмента кода завершается передачей управление на начало другого сегмента. Переменные, присваиваемые в одном сегменте и используемые в другом, должны принадлежать состоянию процесса.

```
СЕГМЕНТ  ::= МЕТКА : [ invariant ИНВАРИАНТ-СЕГМЕНТА ] ТЕЛО-СЕГМЕНТА
ИНВАРИАНТ-СЕГМЕНТА  ::= ФОРМУЛА
```

Когда исполнение процесса достигает начала сегмента, соответствующий ИНВАРИАНТ-СЕГМЕНТА, если он имеется, должен быть истинным.

```
ТЕЛО-СЕГМЕНТА  ::=  КОМПОНЕНТ-СЕГМЕНТА [ | ТЕЛО-СЕГМЕНТА ]
КОМПОНЕНТ-СЕГМЕНТА  ::= ОПЕРАТОР | ВЫЗОВ-ПРОЦЕССА
```

В общем случае сегмент определен в виде недетерминированной композиции компонентов. Результатом исполнения композиции является исполнение одного из компонентов, выбираемого недетерминировано. Исполнение ОПЕРАТОРА всегда завершается либо переходом на начала другого сегмента, либо одним из выходов определяемого процесса.

```
ВЫЗОВ-ПРОЦЕССА ::=
      ИДЕНТИФИКАЦИЯ-ПРОЦЕССА ( [АРГУМЕНТЫ] (: РЕЗУЛЬТАТЫ-ВЕТВИ)* )
РЕЗУЛЬТАТЫ-ВЕТВИ  ::= [РЕЗУЛЬТАТЫ] [ОПЕРАТОР-ПЕРЕХОДА]
```

Процесс, запускаемый на исполнение ВЫЗОВОМ-ПРОЦЕССА, должен иметь соответствующее определение процесса. Отсутствие ОПЕРАТОРА-ПЕРЕХОДА по одной из ветвей означает, что исполнение по данной ветви будет продолжено с начала следующего сегмента.

```
ОПЕРАТОР-ПЕРЕХОДА ::= # МЕТКА
ИДЕНТИФИКАЦИЯ-ПРОЦЕССА ::=
        [ИМЯ-МОДУЛЯ .] ИМЯ-ПРОЦЕССА |
        [ ОБЪЕКТ-КЛАССА . ] ИМЯ-ПРОЦЕССА |
        ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ
```

ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ должно вычислять имя процесса. Если вызываемый процесс является методом некоторого класса, то вызов процесса реализуется через ОБЪЕКТ-КЛАССА.

В теле процесса наряду с операторами, разрешенными в определении предиката, используются дополнительные операторы.

```
ОПЕРАТОР-РАБОТЫ-С-ПРОЦЕССАМИ ::=
        ЗАПУСК-НЕЗАВИСИМОГО-ПРОЦЕССА |
        РЕАКЦИЯ-НА-СООБЩЕНИЯ |
        ПОСЫЛКА-СООБЩЕНИЯ |
        ЗАПУСК-ПАРАЛЛЕЛЬНОГО-ПРОЦЕССА |
        ЗАЩИЩЕННЫЙ-ОПЕРАТОР
```

Взаимодействие между процессами реализуется посредством приема и передачи сообщений.

```
ОПИСАНИЕ-СООБЩЕНИЯ ::=
        message ИМЯ-СООБЩЕНИЯ [( СПИСОК-ТИПОВ )]
ИМЯ-СООБЩЕНИЯ      ::= ИДЕНТИФИКАТОР
СПИСОК-ТИПОВ   ::= ИЗОБРАЖЕНИЕ-ТИПА [, СПИСОК-ТИПОВ]
```

Сообщение содержит (возможно, пустой) набор значений, типы которых указываются в параметрах описания сообщения.

```
ПОСЫЛКА-СООБЩЕНИЯ  ::=
        send ИМЯ-СООБЩЕНИЯ [( СПИСОК-ВЫРАЖЕНИЙ )]
```

Оператор посылает сообщение с набором значений, вычисленных СПИСКОМ-ВЫРАЖЕНИЙ.

```
РЕАКЦИЯ-НА-СООБЩЕНИЯ ::=
       receive СООБЩЕНИЕ ОПЕРАТОР |
       receive СООБЩЕНИЕ { ОПЕРАТОР }  else[ РЕАКЦИЯ-НА-СООБЩЕНИЯ ]
СООБЩЕНИЕ  ::=  ИМЯ-СООБЩЕНИЯ [( СПИСОК-ПЕРЕМЕННЫХ )]
```

Для второго варианта правила при отсутствии первого сообщения в канале проверяются другие сообщения из **else**-части оператора. Оператор реакции на сообщения исполняется многократно до тех пор, пока в канале не появится одно из сообщений, принимаемых оператором реакции на сообщения. После получения сообщения переменные, указанные СПИСКОМ-ПЕРЕМЕННЫХ, получают значения параметров сообщения, и выполняется соответствующий ОПЕРАТОР.

Конструкция СООБЩЕНИЕ в любой позиции логического выражения трактуется как неблокирующий прием сообщений. Исполнение конструкции осуществляется следующим образом. В очереди пришедших сообщений исполняемого процесса ищется сообщение с именем, указанным в конструкции СООБЩЕНИЕ. Если сообщение обнаружено, оно вычеркивается из очереди. При этом значения параметров присваиваются соответствующим переменным, перечисленным в круглых скобках. Результатом исполнения конструкции СООБЩЕНИЕ является значение **true**. Если требуемое сообщение не обнаружено в очереди, исполнение конструкции СООБЩЕНИЕ завершается значением **false**.

```
ЗАПУСК-ПАРАЛЛЕЛЬНОГО-ПРОЦЕССА   ::= spawn ВЫЗОВ-ПРОЦЕССА
```

Реализуется запуск процесса, определяемого ВЫЗОВОМ-ПРОЦЕССА, параллельно с процессом, выполняющим данный оператор.

```
ЗАЩИЩЕННЫЙ-ОПЕРАТОР   ::=
      with ( СПИСОК-РАЗДЕЛЯЕМЫХ-ПЕРЕМЕННЫХ ) ОПЕРАТОР
СПИСОК-РАЗДЕЛЯЕМЫХ-ПЕРЕМЕННЫХ ::= СПИСОК-ПЕРЕМЕННЫХ
```

Взаимодействие между параллельными процессами возможно через разделяемые переменные, доступные для нескольких процессов и являющиеся глобальными по отношению к ним. Доступ к разделяемым переменным разрешен только внутри защищенного оператора. При исполнении ОПЕРАТОРА доступ к перечисленным разделяемым переменным в других процессах блокируется. Блокировка снимается при завершении исполнения ОПЕРАТОРА. Если при попытке исполнить ЗАЩИЩЕННЫЙ-ОПЕРАТОР одна из разделяемых переменных оказалась блокирована другим процессом, исполнение текущего процесса, исполняющего ЗАЩИЩЕННЫЙ-ОПЕРАТОР, приостанавливается до момента снятия блокировки с разделяемой переменной.

Тип **time** используется для переменных и констант, значениями которых являются показания времени. Оператор **set** t, эквивалентный оператору **time** t = 0, реализует установку таймера, переменной t. Ее изменение производится непрерывно некоторым механизмом, не зависящим от автоматной программы. Оператор **delay** T реализует задержку исполнения программы на время T; по истечению этого времени программа продолжит работу со следующего оператора.

12. Императивное расширение

Императивное расширение языка P определяет дополнительные языковые конструкции, возникающие в программе в результате проведения трансформаций предикатной программы; см. введение. Использование этих конструкций в исходной программе недопустимо.

```
ЗАГОЛОВОК-ПРЕДИКАТА ::=
      ( [ОПИСАНИЯ-АРГУМЕНТОВ] [: ОПИСАНИЯ-РЕЗУЛЬТАТОВ] )
ВЫЗОВ-ПРЕДИКАТА-ФУНКЦИИ ::=
      ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА ( [АРГУМЕНТЫ] [: РЕЗУЛЬТАТЫ] )
```

В императивном расширении для предиката-функции допускается отсутствие результатов. Это отражено в приведенных определениях, расширяющие определения, данные в разд. 3.1. Отсутствие результатов возможно, например, после склеивания результатов с глобальными переменными.

Определим дополнительные операторы императивного расширения.

```
ОПЕРАТОР-ИМПЕРАТИВНОГО-РАСШИРЕНИЯ ::=
      break |
      ОПЕРАТОР-FOR |
      ОПЕРАТОР-ЕСЛИ
```

В императивной программе, полученной в результате трансформации, в позиции оператора исходной предикатной программы может также находиться оператор императивного расширения. Отметим, что групповой оператор присваивания, возникающий при подстановке тела предиката на место вызова предиката, является частным случаем оператора присваивания, когда в левой части - мультипеременная, а в правой - мультивыражение.

```
ОПЕРАТОР-FOR  ::=
        for ( ЗАГОЛОВОК-ЦИКЛА ) { ОПЕРАТОР ( ; ОПЕРАТОР )* }
ЗАГОЛОВОК-ЦИКЛА   ::=
        [[ИЗОБРАЖЕНИЕ-ТИПА] ПАРАМЕТР-ЦИКЛА = ВЫРАЖЕНИЕ] ;
              [УСЛОВИЕ-ЗАВЕРШЕНИЯ] ;
              [ПЕРЕСЧЕТ-ПАРАМЕТРА]
ПАРАМЕТР-ЦИКЛА     ::= ИДЕНТИФИКАТОР
УСЛОВИЕ-ЗАВЕРШЕНИЯ ::= ВЫРАЖЕНИЕ
ПЕРЕСЧЕТ-ПАРАМЕТРА ::= ОПЕРАТОР
```

В императивном расширении используются также операторы перехода и помеченные операторы. Синтаксис и семантика определены в разд. 10.

Семантика цикла **for** соответствует языку C++. Выход из цикла **for** может также быть реализован оператором **break** из тела цикла.

```
ЦИКЛ-WHILE ::= while ( ВЫРАЖЕНИЕ ) ОПЕРАТОР
```

Тело цикла **while** исполняется пока логическое выражение в его заголовке истинно.

```
sum (list (int) a : int r) {
  if (len (a) > 0) {
    r = a [0];
    int i = 1;
    while (i < len (a)) {
      r = r + a [i];
      i = i + 1;
    }
  } else {
    r = 0;
  }
}
```

Пример использования цикла while

Данная программа может получиться в результате применения первых трех трансформаций: склеивания переменных, замены хвостовой рекурсии циклом и подстановки определения предиката на место вызова.

Циклы **for** и **while** часто используется для представления циклов, реализуемых с помощью операторов перехода, в целях улучшения структуры программы.

Список литературы

[1] Шелехов В.И. Введение в предикатное программирование. - Новосибирск, 2002. - 82с. - (Препр. / ИСИ СО РАН; N 100).

[2] Шелехов В.И. Язык предикатного программирования Р. - Новосибирск, 2002. - 40с. - (Препр. / ИСИ СО РАН; N 101).

[3] Шелехов В.И. Предикатное программирование: основы, язык, технология. // Методы предикатного программирования / ИСИ СО РАН. - Новосибирск, 2003. - С.7-15.

[4] Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. - Новосибирск, 2004. - 52с. - (Препр. / ИСИ СО РАН; N 115).

[5] Шелехов В.И. Разработка автоматных программ на базе определения требований // Системная информатика, №4, 2014. — ИСИ СО РАН, Новосибирск. — С. 1-29. http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf

[6] Шелехов В.И. Оптимизация автоматных программ методом трансформации требований // «Программная инженерия», №11, 2015. – С. 3-13. http://persons.iis.nsk.su/files/persons/pages/req_k.pdf

[7] PVS Specification and Verification System. - http://pvs.csl.sri.com/

[8] Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. — С. 531–538.

[9] Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, №3, 2014. ИСИ СО РАН, Новосибирск. С. 25-43. [Электронный ресурс] URL: http://persons.iis.nsk.su/files/persons/pages/String.pdf

[10] Тумуров Э.Г., Шелехов В.И. Технология автоматного программирования на примере программы управления лифтом // «Программная инженерия», Том 8, № 3, 2017. – С.99-111. http://persons.iis.nsk.su/files/persons/pages/lift1.pdf

[11] Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р — Новосибирск, 2010. — 42с. — (Препр. / ИСИ СО РАН; N 153).

[12] Шелехов В. И., Тумуров Э. Г. Методы автоматного программирования для разработки и верификации систем управления // Программная инженерия. 2024. Том 15, №2. С. 73-86. https://persons.iis.nsk.su/files/persons/pages/aubridge.pdf

[13] Шелехов В.И., Каблуков И.В., Янбулатов Д.Р. Разработка и верификация программ обработки графов в предикатном программировании // Знания-Онтологии-Теории (ЗОНТ-23). — Новосибирск, ИМ СО РАН, 2023. — С. 299-306. https://persons.iis.nsk.su/files/persons/pages/shelekhov-zont23.pdf

Приложение А

Библиотека программ для строковых объектов

Для объектов типа string не определено отношение равенства «==». Для лексикографического сравнения строк используется функция Compare с результатом 0 при совпадении строк, -1 – если первая строка меньше второй и 1 – если первая строка больше второй.

```
Compare(string s, t: int ) {
    char c = s.car, d = t.car;
    if (c == d) {
        if (c == 0) return 0 else return Compare(s.cdr, t.cdr)
    } else return c - d
}
```

Программа SubString заменяет строку s ее вырезкой длины n начиная с элемента по номеру p, т.е. вырезкой s[p..p + n]. При этом итоговая вырезка не может выходить за границу исходной строки s.

```
SubString(string s, nat p, n: string s'){
    if (p >= length(s)) s' = empty
    else s' = s[p..min(p + n, length(s) - 1)];
}
```

При p≠0 итоговое значение s' получается сдвигом исходного значения s в памяти. Не следует использовать данную программу в случае, когда значение переменной s' далее не модифицируется. Предпочтительней использовать операции вырезки, которая будет реализована через объект сканирования.

Программа Insert вставляет строку t внутрь строки s начиная с позиции p.

```
Insert(string s, t, nat p: string s'){
    if (p > length(s) or t == empty) return;
    nat L = length(s) + length(t);
    if (L > max_len(s)) resize(s, L);
    if (p = 0) { s' = t + s; return };
    s' = s[0..p-1] + t + s[p..]
}
```

Эффективная реализация оператора s' = s[0..p-1] + t + s[p..] в императивном расширении обеспечивается парой вызовов программ из внутренней библиотеки:

```
right(s, p, length(s), p + length(t));
copy(s, t, p)
```

Вызов right сдвигает вырезку s[p..] вправо (вместе с завершающим нулем) на length(t) позиций. Вызов copy переписывает значение строки t в строку s начиная с позиции p.

Программа Replace заменяет часть строки, соответствующей вырезке s[p..p + n], на строку t. Если p + n выходит за границы строки s, заменяемая вырезка ограничена концом строки.

```
Replace(string s, t, nat p, n: string s'){
    if (p > length(s)) return;
    if (p = length(s)) { s' = s + t; return };
    nat m = (p + n >= length(s))? length(s) – 1 : p + n;
    nat L = length(s) + length(t) – m + p – 1;
    if (L > max_len(s)) resize(s, L);
    s' = s[0..p-1] + t + s[m+1..]
}
```

Функция StartsWith проверяет, является ли строка t начальной частью строки s.

```
StartsWith(string s, t: bool ) post ∃ string u. s = t + u {
    if (t.car == 0) return true;
    if (s.car != t.car) return false;
    return StartsWith(s.cdr, t.cdr)
};
```

Функция EndsWith проверяет, является ли строка t конечной частью строки s.

```
EndsWith(string s, t: bool ) {
    return EndsWith(string s, t, length(s), length(t));
} post ∃ string u. s = u + t;

EndsWith(string s, t, nat js, jt: bool ) {
    if (s[js] != t[jt]) return false;
    if (jt == 0) return true;
    if (js == 0) return false;
    return StartsWith(s, t, js - 1, jt - 1)
};
```

Гиперфункция Index определяет первую позицию элемента c в строке s начиная с позиции p. Результат – позиция q. При отсутствии элемента c реализуется вторая ветвь #noel.

```
Index(string s, char c, nat p: nat q #yes : #noel){
    if (p >= length(s)) #noel;
    Index1(s, c, p: q #yes : #noel)
}

Index1(string s, char c, nat q: nat q' #yes: #noel){
    if (s[q] == c) {q' = q; #yes };
    if (s[q] == 0) #noel;
    Index1(s, c, q+1: q' #yes : #noel)
}
```

Гиперфункция RIndex определяет последнюю позицию элемента c в строке s не превышающую позиции p. Результат – позиция q. При отсутствии элемента c реализуется вторая ветвь гиперфункции #noel.

```
RIndex(string s, char c, nat p: nat q #yes: #noel){
    if (p>= length(s)) p = length(s) – 1;
    RIndex1(s, c, p: q #yes: #noel)
}

RIndex1(string s, char c, nat q: nat q' #yes : #noel){
    if (s[q] == c) {q' = q; #yes };
    if (q == 0) #noel;
    RIndex1(s, c, q – 1: q' #yes : #noel)
}
```

Гиперфункция Index определяет позицию первого вхождения строки t в строке s начиная с позиции p. Результат – позиция q. При отсутствии вхождений, а также в случае пустой строки t реализуется вторая ветвь гиперфункции #nostr.

```
Index(string s, t, nat p: nat q #yes: #nostr){
    if (t == empty) #nostr;
    Index1(s, t, p, length(s) – length(t) : q #yes : #nostr) }
```

```
Index1(string s, t, nat p, Lts: nat q #yes : #nostr){
    if (p > Lts) #nostr;
    Index2(s, t, p, 0 : q #yes : )
    Index1(s, t, p+1, Lts: q #yes : #nostr)
}

Index2(string s, t, nat p, j: nat q #yes : #no){
    if (t[j] == 0) {q = p; #yes};
    if (t[j] != s[p+j]) #no;
    Index2(s, t, p, j+1: q #yes : #no)
}
```

После трансформаций замены рекурсии циклом и подстановки тел программ на место вызовов программа Index приводится к следующему виду:

```
Index(string s, t, nat p: nat q #yes: #nostr){
    if (t == empty) #nostr;
    nat Lts = length(s) – length(t);
    for (; ; p = p+1) {
        if (p > Lts) #nostr;
        for (j=0; ; j = j+1) {
            if (t[j] == 0) {q = p; #yes};
            if (t[j] != s[p+j]) break;
        }
    }}
```

Гиперфункция RIndex определяет позицию последнего вхождения строки t в строке s не превышающую позиции p. Результат – позиция q. При отсутствии вхождений, а также в случае пустой строки t реализуется вторая ветвь гиперфункции #nostr.

```
RIndex(string s, t, nat p: nat q #yes : #nostr){
    if (t == empty) #nostr;
    if (p>= length(s)) p = length(s) – 1;
    RIndex1(s, t, p, length(t) – 1: q #yes : #nostr)
}

RIndex1(string s, t, nat p, Lt1: nat q #yes : #nostr){
    if (Lt1 > p) #nostr;
    RIndex2(s, t, p, Lt1 : q #yes : )
    RIndex1(s, t, p – 1, Lt1: q #yes: #nostr)
}

RIndex2(string s, t, nat q, j: nat q’ #yes : #no){
    if (t[j] != s[q]) #no;
    if (j == 0) {q’ = q; #yes};
    RIndex2(s, t, q – 1, j – 1: q’ #yes : #no)
}
```

Гиперфункция FirstOf определяет позицию первого вхождения любого элемента строки t в строке s начиная с позиции p. Результат – позиция q. При отсутствии вхождений, а также в случае пустой строки t реализуется вторая ветвь гиперфункции #noel.

```
FirstOf(string s, t, nat p: nat q #yes: #noel) {
    if (t == empty) #noel;
    FirstOf1(s, t, p, length(s) : q #yes : #noel)
}

FirstOf1(string s, t, nat p, Ls: nat q #yes : #nostr){
    if (p >= Ls) #nostr;
    FirstOf2(s, t, p, 0 : q #yes : )
    FirstOf1(s, t, p+1, Ls: q #yes : #nostr)
}

FirstOf2(string s, t, nat p, j: nat q #yes : #no){
    if (t[j] = s[p]) {q = p; #yes};
    if (t[j] == 0) #no;
    FirstOf2(s, t, p, j+1: q #yes : #no)
}
```

13. Дополнения к языку P

13.1. Директивы компилятора

```
ДИРЕКТИВЫ-КОМПИЛЯТОРА ::=
    pragma ДИРЕКТИВА-КОМПИЛЯТОРА (, ДИРЕКТИВА-КОМПИЛЯТОРА)* [БЛОК]
ДИРЕКТИВА-КОМПИЛЯТОРА    ::=
    ИМЯ-ДИРЕКТИВЫ [( ПАРАМЕТРЫ-ДИРЕКТИВЫ-КОМПИЛЯТОРА )]
ПАРАМЕТРЫ-ДИРЕКТИВЫ-КОМПИЛЯТОРА ::=
    ПАРАМЕТР-ДИРЕКТИВЫ-КОМПИЛЯТОРА
        (, ПАРАМЕТР-ДИРЕКТИВЫ-КОМПИЛЯТОРА)*
ПАРАМЕТР-ДИРЕКТИВЫ-КОМПИЛЯТОРА  ::=
     КЛЮЧЕВОЕ-СЛОВО |
     ИДЕНТИФИКАТОР |
     КОНСТАНТА |
     МЕТКА
```

Директивы могут относиться ко всему файлу, в котором объявлены, к его части, либо к некоторой части исходного кода внутри предиката.

Директивы могут объявляться вне предикатов либо внутри предикатов. Если директивы объявлены вне какого предиката, их действие распространяется с места определения до места переопределения либо до конца файла, если они не переопределяются, при этом БЛОК за их определением должен отсутствовать. Если директивы объявлены внутри предиката, в их определении должен присутствовать БЛОК, на который распространяется их действие.

Определим следующие директивы компилятора: int_bitness (РАЗРЯДНОСТЬ-ЦЕЛЫХ), real_bitness (РАЗРЯДНОСТЬ-ВЕЩЕСТВЕННЫХ), int_overflow (ПЕРЕПОЛНЕНИЕ-ЦЕЛЫХ) и real_overflow (ПЕРЕПОЛНЕНИЕ-ВЕЩЕСТВЕННЫХ).

```
ИМЯ-ДИРЕКТИВЫ        ::= int_bitness | real_bitness | int_overflow | real_overflow
РАЗРЯДНОСТЬ-ЦЕЛЫХ        ::= 1 | 2 | ... | 64 | native | unbounded
РАЗРЯДНОСТЬ-ВЕЩЕСТВЕННЫХ  ::= 32 | 64 | 128 | unbounded
ПЕРЕПОЛНЕНИЕ-ЦЕЛЫХ       ::= wrap | strict | fail | # МЕТКА
ПЕРЕПОЛНЕНИЕ-ВЕЩЕСТВЕННЫХ ::= safe | strict | fail | # МЕТКА
```

В дальнейшем допускается добавления других директив.

Битности целых и вещественных относятся к типам int, nat и real, объявленным без параметров. По умолчанию они не ограничены.

int_overflow и real_overflow управляют поведением в случае переполнения. Они могут иметь следующие параметры:

strict используется для Семантика цикла **for** соответствует языку C++. Выход из цикла **for** может также быть реализован оператором **break** из тела цикла. означает проверку на переполнение на этапе компиляции, если не удаётся определить, возможно ли переполнение, либо установлен факт возможности переполнения, то возникает ошибка компиляции.

fail вызывает проверку времени исполнения, при этом, если происходит переполнение, исполнение останавливается и печатается отладочная информация, которая может помочь программисту выявить и исправить ошибку в программе.

wrap означает использование специальных арифметических операций для целых чисел, например, сложение можно определить предикатом, эквивалентным следующему:

```
addition (type t, t a, t b : t r) {
    r = (a + b - low (t)) % (high (t) - low (t)) + low (t);
}
```

где на место high (t) и low (t) во время компиляции подставляются минимальное и максимальное значение типа t (будем считать, для определённости, что внутри этой операции вычисления выполняются с произвольной точностью и переполнения не происходит).

safe так же означает использование альтернативных операций: если результат операции лежит между минимальным и максимальным значением вещественного типа, он приводится к ближайшему значению, представимому с заданной точностью; если результат больше максимального значения, будем считать его равным inf (бесконечности); если же результат меньше минимально допустимого значения, будем считать его равным -inf.

При локальном объявлении (т.е., внутри предиката) директивы компилятора int_overflow или real_overflow допускается использование в качестве параметра метки ветви предиката. В этом случае при переполнении исполнение предиката завершается со срабатыванием соответствующей ветви. Например:

```
safe_sum (int (32) a, int (32) b : int (32) result #ok: #err) {
  pragma int_overflow (#err) { result = a + b; };
}
// ...
safe_sum (a, b : int (32) r #ok: #err)
  case ok:  print ("a + b = ", r)
  case err: print ("Произошло переполнение!");
```

Предложенный выше подход позволяет не вдаваться в детали реализации там, где этого не требуется, и при этом сохранить полный контроль над машинным представлением чисел в случаях, когда это необходимо.

13.2. Использование * при описании параметров определения предиката

```
ОПИСАНИЯ-ГРУППЫ-АРГУМЕНТОВ  ::=
        ИЗОБРАЖЕНИЕ-ТИПА ИМЯ-АРГУМЕНТА (, ИМЯ-АРГУМЕНТА)* |
        ИЗОБРАЖЕНИЕ-ТИПА-КАК-ПАРАМЕТРА
ОПИСАНИЯ-АРГУМЕНТОВ   ::=
        ОПИСАНИЯ-ГРУППЫ-АРГУМЕНТОВ [, ОПИСАНИЯ-АРГУМЕНТОВ]
ИМЯ-АРГУМЕНТА     ::= ИДЕНТИФИКАТОР | ИДЕНТИФИКАТОР *
```

Использование аргумента вида имя* определяет аргумент имя и результат имя’. При этом результат имя’ не должен встречаться в описании результатов предиката. Переменная имя’ считается находящейся в начале списка результатов предиката. Если имеется несколько аргументов, снабженных звездочкой *****, то порядок соответствующих результирующих переменных со штрихом **’** в списке результатов соответствует порядку аргументов со звездочкой.

```
assign (int from : int to) { to = from }
main (seq (string) argv : int ret_code) { ret_code = 0 }
power (real x*, int p :) { x’ = x^p; }
```

Примеры определений предикатов

```
ОПРЕДЕЛЕНИЕ-ПРЕДИКАТА-ГИПЕРФУНКЦИИ ::=
        ЗАГОЛОВОК-ПРЕДИКАТА-ГИПЕРФУНКЦИИ [ПРЕДУСЛОВИЯ-ГИПЕРФУНКЦИИ]
            ТЕЛО-ПРЕДИКАТА [ПОСТУСЛОВИЯ-ВЕТВЕЙ]
ЗАГОЛОВОК-ПРЕДИКАТА-ГИПЕРФУНКЦИИ   ::=
        ИМЯ-ПРЕДИКАТА ( [ОПИСАНИЯ-АРГУМЕНТОВ] :
                    ОПИСАНИЯ-РЕЗУЛЬТАТОВ-ГИПЕРФУНКЦИИ )
```

В ОПИСАНИИ-АРГУМЕНТОВ предиката-гиперфункции не допускается звездочка ***** в конце имени параметра-аргумента.

13.3. Предикат комбинирования

Для структурных выражений определён предикат комбинирования, с помощью которой можно сопоставлять значение структуры некоторому произвольному значению определённым образом. Наиболее распространённым примером этому может служить форматирование строк, отображающее строку, в которой задаётся формат, и кортеж в отформатированную заданным образом строку. Рассмотрим пример:

```
type format_value_t = union (int d, real f, string s);

fmtPart (format_value_t v, string fmt, string s, unsigned int fieldNum : s') {
  // v:        значение, в виде объединения поддерживаемых типов,
  // fmt:      строка форматирования,
  // s:        уже отформатированная часть строки,
  // fieldNum: порядковый номер обрабатываемого поля структуры (по этому номеру
  //           получаем правила форматирования для данного значения)
}

format (string fmt, type structType, structType a : string s)
{ combine (a, fmt, "", fmtPart : s); }

string date = format ("%02d.%02d.%d", struct (int, int, int), (26, 2, 2008));
                                                          // date = "26.02.2008"
```

Пример использования combine

В общем случае, первым параметром combine может быть произвольная структура, вторым — выражение произвольного типа, типы выходного и третьего входного параметра должны совпадать, а четвёртым входным должен быть предикат, со следующими ограничениями на параметры:

- Первый параметр – объединение, такое что для каждого типа, участвующего в кортеже, являющимся первым параметром combine, в нём существует единственная альтернатива с таким типом. При этом метка альтернативы может быть произвольной, компилятор выберет альтернативу, основываясь на уникальности типов, а не на метке.
- Второй параметр это в точности второй параметр combine. Любое значение, переданное в combine, будет передано и в этот предикат.
- Тип третьего параметра должен совпадать с типом третьего параметра combine, а так же выходных параметров combine и этого предиката. При первом вызове этого предиката значение данного

параметра равно значению соответствующего параметра combine. При последующих вызовах оно равно результату предыдущего вызова этого предиката.

- Четвёртым параметром передаётся порядковый номер вызова предиката (считая с нуля). Предикат вызывается последовательно для каждого поля входной структуры. Результат последнего вызова возвращается как результат combine.

Конечно, предикат комбинирования вводится прежде всего именно чтобы обеспечить возможности реализации форматирования строк, но также очевидно, что для него существуют и другие применения.

13.4. Специальная семантика полей типа структуры

При определении полей структуры идентификатор может не указываться, в этом случае доступ к этому полю производится по индексу или с помощью базисных предикатов head и tail. Если два определения структур отличаются только идентификаторами полей, то они определяют один и тот же тип.

```
type person_t = struct (
   string familyName,
   string givenName,
   nat age
);
person_t somePerson = ("Smith", "John", 21);
person_t someOtherPerson = (givenName: "Jane", familyName: "Doe", age: 18);
nat n = somePerson.age;
```

Пример использования полей структур

Если при определении константы структурного типа используются идентификаторы полей (см. строку 6 примера), то такая константа совместима также и с теми структурами, у которых совпадают типы и идентификаторы полей, т.е., инициализация в строке 7 примера корректна.

13.5. Базисные предикаты с расширенной функциональностью

Таблица 3 Базисные предикаты

	Описание	array	list	map	seq	set	struct
len	Количество элементов	+	+	+	+	+	+
empty	Проверка на пустоту	+	+	+	+	+	+
head	Первый элемент	+	+	-	+	-	+

tail	Элементы, следующие за первым	+	+	-	+	-	+
keys	Множество ключей ассоциативного массива	-	-	+	-	-	-
dim	Число измерений массива (0, 1 или 2)	+	-	-	+	-	-
combine	Комбинирование полей структуры	-	-	-	-	-	+

13.6. Дополнительная семантика массивов

Массивы одинакового размера и одинаковым типом элементов совместимы, т.е. выражения одного из этих типов могут неявно приводиться компилятором к другому из этих типов (иначе операторы, вроде определения переменной в строке 4 примера ниже, выглядели бы нелогично из-за возможного несовпадения типов).

Допустимо использование символов **"..."** вместо описания элементов массива. В этом случае, конкретная размерность массива привязывается к переменной такого типа при инициализации (см. строку 1 примера). Это позволяет использовать одни и те же предикаты для работы с массивами произвольного размера.

```
type int_mtx_t = array (int, ..., ...);
int_mtx_t m = [ [ 1, 2, 3 ],
                [ 4, 5, 6 ] ];
int size    = len (m);  // size = 6
int dim     = dim (m);  // dim  = 2
int e = m [1, 2]; // e = 6
```

Пример операций над массивами

С массивами можно использовать предикат len, возвращающий количество элементов, и dim, возвращающий количество измерений. Также для них определена операция +, возвращающая объединение массивов-операндов (в случае, если их типы элементов совпадают, а множества индексов являются подтипами одного типа и не пересекаются) и логическая операция in, возвращающая истину, если её левый операнд является элементом массива (правый операнд), и ложь иначе.

13.7. Ассоциативные массивы

```
ОПРЕДЕЛЕНИЕ-АССОЦИАТИВНОГО-МАССИВА ::=
        map ( ИЗОБРАЖЕНИЕ-ТИПА , ИЗОБРАЖЕНИЕ-ТИПА )
КОНСТРУКТОР-АССОЦИАТИВНОГО-МАССИВА ::=
        [{ ( ВЫРАЖЕНИЕ : ВЫРАЖЕНИЕ )* }]
```

```
type string_map_t = map (string, string);
string_map_t m    = [{ "qwerty" : "asdf", "foo" : "bar" }];
string s          = m ["foo"]; // s = "bar"
set (string) k = keys (m); // k = { "qwerty", "foo" }
```

Пример использования ассоциативного массива

Ассоциативные массивы позволяют ставить в соответствие значения произвольных типов.

Аналогично прочим массивам, для ассоциативных массивов определены операции len, in, +, с той разницей, что in проверяет наличие индекса, а не наличие элемента. Также определён предикат keys, возвращающий множество индексов.

13.8. Представления списков для эффективной реализации

Основным способом представлением списка является массив. В качестве возможных альтернатив рассматриваются другие способы в виде: односвязного списка, двусвязного списка. Представление в виде кольцевого буфера следует считать модификацией представления в виде массива.

Далее рассматривается лишь представление списка в виде массива [9].

Для изображения типа списка допускается использование следующих типовых термов:

```
list(T)
list(T, L)
```

Здесь T – тип элемента списка, L – максимальная длина списка. Размер памяти, отводимой для переменной типа list(T, L), будет достаточным для размещения L элементов списка.

Допустим, отведенная для списковой переменной память есть массив A с индексами в диапазоне 0..N. Тогда значение списковой переменной можно представить вырезкой A[m..n], где $0 \le m \le n \le N$, однако в случае пустого списка $m > n$. В большинстве случаев $m = 0$ и свободное место в памяти остается слева в диапазоне m+1..N. Однако бывают случаи, когда свободную часть памяти надо оставить справа для того, чтобы реализовать присваивание вида s = y + s, как, например, в работе [4], где используется присваивание buf = stf + buf. Для формирования нестандартного размещения значения списка используется специальный конструктор.

```
СПЕЦИАЛЬНЫЙ–КОНСТРУКТОР-СПИСКА ::=
        consLeft ( ВЫРАЖЕНИЕ-ТИПА-СПИСОК , ВЫРАЖЕНИЕ ) |
        consRight (ВЫРАЖЕНИЕ-ТИПА-СПИСОК ) |
        consRight (ВЫРАЖЕНИЕ-ТИПА-СПИСОК , ВЫРАЖЕНИЕ )
ВЫРАЖЕНИЕ-ТИПА-СПИСОК ::= ВЫРАЖЕНИЕ
```

Конструктор вида consLeft(y, m), где m – номер элемента, формирует представление списка y в массиве, сдвинутое на m элементов относительно начала массива. Конструктор вида consRight(y) формирует представление списка y в массиве прижатым вправо. Конструктор вида consRight(y, L) формирует представление списка y в массиве длины L прижатым вправо. При исполнении оператора s = consRight(y, L) отводится новая память для строковой переменной s; если до присваивания переменная s уже имела некоторое значение, то транслятор с языка P должен обеспечить возврат старой памяти переменной s.

В языке P нет операторов отведения и освобождение памяти для списковых переменных. Вставка в код соответствующих действий реализуется транслятором. Отведение памяти реализуется

при первом присваивании переменной. Размер памяти определяется по значению правой части оператора присваивания, если он явно не указан описанием типа.

Для значения списковой переменной не допускается выход значения за границы памяти, отведенной для переменной. Соответствующий контроль возлагается на программиста. Для этой цели предусмотрены следующие конструкции.

```
max_len(s)
store(s)
left_store(s)
resize(s, n)
```

Здесь s – переменная типа список. Значением функции max_len(s) является максимальное число элементов списка, которое можно разместить в массиве для переменной s. Функция store(s) определяет число элементов, которое можно разместить справа от значения s в свободной части памяти. Функция left_store(s) определяет число элементов, которое можно разместить слева от значения s в памяти. Оператор resize(s, n) отводит новую память размера n элементов, переписывает туда значение переменной s, освобождая старую память.

13.9. Оптимизация памяти для строк

Следует сохранить возможность указания размера памяти для строковых объектов. В качестве альтернативного способа изображения строкового типа предлагается использовать string(L): память для значения типа string(L) вмещает ровно L литер. Полезны также конструкции, введенные для списков:

```
max_len(s)
store(s)
resize(s, n)
```

Исключается возможность сдвига вправо значения строки относительно начала памяти, т.е значение строки всегда размещается с начала памяти.

Целесообразно использовать два вида объектов сканирования: один – для сканирования с начала строки, второй – с конца. В первом случае объект сканирования может быть представлен указателем на начальный элемент строки, во втором – дополнительно требуется длина строки, при этом строка, представленная объектом сканирования, нулем не завершается.

Изменения версии 0.14

19 июня 2018г.

1. Постусловие переносится перед телом предикатной программы.

2. Вводится описание меры рекурсивной программы и рекурсивно определяемых формул.

3. Из работы [9] введены расширения для списков и строк: вырезка списка, специальные конструкторы, операции управления памятью, нуль-терминированное представление для строковых.

4. Приложение А. Библиотека программ для строковых объектов. Адаптирована из работы [9].

5. Переписано введение.

6. Конструкция **context** для набора внешних аргументов, которые можно опускать в вызовах программ и формул, а также в типовых термах.

7. Вводится ОПИСАНИЕ-ТЕОРИИ, определяющее набор УТВЕРЖДЕНИЙ.

8. В качестве кванторов запрещается использовать **!** и **?.**

9. Внешние аргументы предикатных программ, разд. 3.1, 3.3, 7, 9.

10. Расширен условный оператор. Используется **elsif**. Разд. 5

11. Мульти-переменные и мульти-выражения можно будет использовать только в операторе мульти-присваивания. Разд. 5

12. В описании подтипа вместо логического ВЫРАЖЕНИЯ теперь ФОРМУЛА. Разд. 7

Изменения версии 0.12

3 декабря 2013г.

1. Изменены описания классов. Теперь они в конце разд. 6. Переработан разд. 10, определяющий описания процессов. Все изменения отмечены тремя плюсами +++
2. Разд. 4. Аксиомы, леммы и теоремы определяются конструкцией УТВЕРЖДЕНИЕ, одной из альтернатив ОПИСАНИЯ.
3. Разд. 3.3. В позиции вызова предиката должно быть ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ, а не ВЫРАЖЕНИЕ
4. Разд.6. Введено ВТОРИЧНОЕ-ВЫРАЖЕНИЕ в целях ограничить ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ в конструкциях типа “поле структуры”.

Изменения версии 0.11

1. Разд. 10. Аксиомы, леммы и теоремы определяются конструкцией УТВЕРЖДЕНИЕ, одной из альтернатив ОПИСАНИЯ.
2. Разд. 10. Введена конструкция ТЕОРИЯ. Она может встречаться в составе описаний модуля программы. УТВЕРЖДЕНИЯ, должны входить в состав некоторой ТЕОРИИ, и не может встречаться в программе вне ТЕОРИИ.
3. Разд. 10. Откат назад на один шаг. Конструкция ТЕОРИЯ устраняется из языка. Конструкция УТВЕРЖДЕНИЕ является одной из альтернатив ОПИСАНИЯ.

Изменения версии 0.10

1. Разд. 4. 9. ОПИСАНИЕ-ПРЕДИКАТА-СПЕЦИФИКАЦИИ заменено на ОПИСАНИЕ-ФОРМУЛЫ. Ранее формула определяла предикат. Сейчас - функцию, тип которой указывается после двоеточия. Если тип - **bool**, то формула определяет предикат.

2. Разд. 5. Из правила ОПРЕДЕЛЕНИЕ-ЛОКАЛЬНОГО-ПРЕДИКАТА изъято (удалено) ОПРЕДЕЛЕНИЕ-СУЖЕННОГО-ПРЕДИКАТА. Эта форма оказалась избыточной.

3. Разд. 6. В конструкцию МОДИФИКАЦИЯ добавлен разделитель **with**.

4. Разд. 4. Введены формальные параметры модулей и фактические параметры при импорте модулей. Импорт может оказаться не в начале модуля.

5. Из агрегатов исключены объединения.

6. Разд. 7. Тип объединения union теперь имеет структуру алгебраического типа. Альтернатива объединения - конструктор с аргументами, называемыми полями объединения. Тип перечисления определен как частный случай типа объединения.

7. Термин “последовательность” везде заменяется на “список”. Описатель **seq** заменяется на list. Вместо функций head и tall используются поля car и cdr. Материал из дополнений к языку, разд. 12.7, определяющий списки, частично переносится в разд. 6 и 7. Вводится констркуция АГРЕГАТ-СПИСОК. Пустой агрегат теперь не представляется константой **nil**.

8. Разд. 7. Удалено ИЗОБРАЖЕНИЕ-ТИПА-С-ПУСТЫМ-ЗНАЧЕНИЕМ.

9. Разд. 6. Префиксом для элемента массива, поля структуры и поля класса является ПЕРВИЧНОЕ-ВЫРАЖЕНИЕ, а не ВЫРАЖЕНИЕ

10. Разд. 5. Удалено предложение: Оператор выбора должен быть полным: при отсутствии части **default** набор значений кодов должен покрывать набор возможных значений выражения в заголовке. — есть случаи, когда оно может не выполняться

11. Разд. 7. Исправлены примеры программ со списками (листинги 7, 8, 10, 13)

12. Разд. 7. Удалено ИЗОБРАЖЕНИЕ-ТИПА-С-ПУСТЫМ-ЗНАЧЕНИЕМ

13. Разд. 11. Подраздел “Деректривы компилятора” пока перемещен в Дополнения к языку. Эта часть требует серьезной доработки, и сейчас в публикации ее лучше не показывать.

14. Переставлены разделы 9 и 10.

15. Разд. 11. Удален описатель mutable, поскольку пользователь не может программировать непосредственно на императивном расширении.

16. Разд. 4. Изменен синтаксис клнструкцмм ИМПОРТ-МОДУЛЯ

17. Разд. 5. Изменен синтаксис. Описания переменных могут находиться перед любым оператором. Сделаны уточнения синтаксиса условного и параллельного операторов.

18. Разд. 4. Вместо printf языка C++ используется собственный оператор print

19. Разд. 7, 8.3. Тип может отсутствовать при описание переменных-полей конструктора в альтернативе **case** оператора **switch** и переменных - индексов массива в операторе **for**.

Изменения версии 0.9

1. Вместо оператора КОНСТРУКТОР-МАССИВА теперь используется операция ОПРЕДЕЛЕНИЕ-МАССИВА.

2. Вместо **lambda** используется **predicate**.

3. Разд. 8.4. Описана семантика операции объединения масивов.

4. Разд. 6. Упрощена конструкция АГРЕГАТА. Вместо ПРИСОЕДИННОГО-АГРЕГАТА используется операция МОДИФИКАЦИЯ.

5. Введены конструкции ОПИСАНИЕ-ПРЕДИКАТА и ОПИСАНИЕ-СПЕЦИФИКАЦИИ-ПРЕДИКАТА. Упрощается определение генратора предиката и изображения предикатного типа.

6. Разд. 9. Изменения в определении ФОРМУЛЫ. Введено описание предиката спецификации. Введены импликация и логическое тождество.

7. Разд. 9. Добавлен ограничитель **formula** при описании предиката спецификации. Вместо <=> используется просто =

8. Разд. 5. В одной альтернативе оператора выбора допускается несколько кодов альтернатив.

9. Разд. 8.2, 8.3. Исправлен синтаксис ИНДЕКСА-ЧАСТИ и др. связанных понятий

10. Разд. 9. Обощено понятие СПИСОК-ПОДКВАНТОРНЫХ-ПЕРЕМЕННЫХ

11. Разд. 10. В защищенном операторе изменен синтаксис

12. Разд. 11. В императивном расширении удален групповой оператор присваивания, поскольку это частный случай оператора присваивания.

13. Разд. 7. Уточнение семантики: изображение диапазона является подмножеством типа **int**

14. Разд. 7. Введены операции **last** и **pred** для последовательностей.

15. Разд. 4, 8.3. Для описания переменной возможно использование описателя **var**.

16. Разд. 9. Кванторы могут также изображаться ограничителями **forall** и **exists**.

17. Разд. 6. Для мультипеременной и мультивыражения допускается опускать скобки “|”

18. Разд. 3.3. Описатель типа локала в позиции результата вызова относится только к этому локалу и не распространяется на последующие результаты.

19. Разд. 5. В операторной позиции может находиться определение локального предиката. Частным случаем является суженный предикат.

20. Разд. 7. Вместо **pred** теперь используется **prec.**

21. Разд. 7. Базисными типами при изображении диапазона могут быть **int**, **enum**, или **char**