АКТОРНЫЙ ПРОЛОГ

ОПРЕДЕЛЕНИЕ

ЯЗЫКА ПРОГРАММИРОВАНИЯ

ВЕРСИЯ ОТ 23 ЯНВАРЯ 2004 ГОДА

А. А. Морозов, Ю. В. Обухов

Áкторный Пролог — объектно-ориентированный логический язык, предназначенный для программирования информационных систем, функционирующих в динамическом внешнем окружении (интеллектуальных агентов Интернет, систем интерактивного проектирования и др.).

Акторный Пролог воплощает новый подход к объединению логического и объектно-ориентированного программирования, обладающий следующими достоинствами:

- В основе нашего подхода лежит использование классической логики (логики предикатов первого порядка).
- Центральной идеей и сущностью подхода является обнаружение и устранение логических противоречий, возникающих в процессе взаимодействия объектов.
- Разработанный подход позволил математически корректным образом ввести в логический язык разрушающее присваивание и параллельные процессы.

Данная публикация является определением синтаксиса и семантики Акторного Пролога (версия от 23 января 2004 г.)

A. A. Morozov, Yu. V. Obukhov

Actor Prolog

Programming Language Definition

(version of January 23, 2004)

Оглавление

Введение

Áкторный Пролог — объектно-ориентированный логический язык, предназначенный для программирования информационных систем, функционирующих в динамическом внешнем окружении (интеллектуальных агентов Интернет, систем интерактивного проектирования и др.).

В Акторном Прологе информационная система представляется в виде теоремы на логическом языке, разделённой на «логические акторы» — повторно доказываемые подцели, взаимодействующие через общие переменные. Доказательство логических акторов (далее — просто «акторов») осуществляется в объектно-ориентированном пространстве поиска, топология которого соответствует структуре системы.

Акторный Пролог воплощает новый подход к объединению логического и объектно-ориентированного программирования, обладающий следующими достоинствами:

1. В основе нашего подхода лежит использование классической логики (логики предикатов первого порядка).

2. Центральной идеей и сущностью подхода является обнаружение и устранение логических противоречий, возникающих в процессе взаимодействия объектов.

3. Разработанный подход позволил математически корректным образом ввести в логический язык разрушающее присваивание и параллельные процессы.

В Акторном Прологе внешние воздействия, вызывающие отклик информационной системы, интерпретируются как использование разрушающего присваивания, вызывающего повторное доказательство акторов логической программы. В частности, взаимодействие человека и машины рассматривается как доказательство некоторой теоремы, в котором одновременно при-

нимают участие человек, изменяющий исходные данные, и машина, обеспечивающая корректность и полноту доказательства.

Стратегия управления Акторного Пролога — акторный механизм — является расширением стандартной стратегии управления языка Пролог (стратегии «поиска слева направо в глубину с возвратом»). В отличие от стандартной стратегии управления, акторный механизм допускает разрушающее присваивание значений общим переменным и при этом автоматически поддерживает корректность доказательства теоремы с помощью повторного доказательства отдельных акторов.

Все синтаксические конструкции Акторного Пролога, за исключением чётко очерченного набора (внелогических) управляющих операторов, имеют классическую декларативную (теоретико-модельную) семантику и могут быть однозначно представлены в виде формул логики предикатов первого порядка.

Различаются «минимальная», «быстрая» и «максимальная» версии языка. Минимальной и быстрой версиям соответствует упрощённый алгоритм унификации (без проверки вхождения). В быстрой и максимальной версиях гарантируется, что исполнение любого процесса не может привести к неопределённо длительной приостановке других процессов.

Для определения синтаксиса языка используется расширенная форма Бэкуса-Наура. Терминальные символы, когда это необходимо, выделяются с помощью кавычек и апострофов.

Настоящий раздел (введение), а также любые приложения, примеры, примечания и ссылки не являются составной частью определения языка.

Ссылки: актор 7.1, акторный механизм 7, доказательство актора 6.3.1, значение переменной 3.1, исполнение процесса 5.2, общие переменные 7.2, повторные доказательства 7.1, подцель доказательства 6.3.1, проверка вхождения 3.3, программа 4, процесс 5.2, разрушающее присваивание 8.1, унификация 3.3.

Глава 1

Алфавит языка

В качестве алфавита языка используется набор символов ASCII, при этом различаются графические символы (графемы), имеющие визуальное представление в виде отпечатанного знака или пробела, и управляющие символы: возврат на одну позицию, горизонтальная табуляция, перевод строки, вертикальная табуляция, перевод формата и возврат каретки. Минимальный набор графических символов, достаточный для определения языка, включает буквы, цифры, символ пробела и специальные символы.

буква = большая_буква | маленькая_буква
большая_буква =
"A" | "B" | "C" | "D" | "E" | "F" | "G" |
"H" | "I" | "J" | "K" | "L" | "M" | "N" |
"O" | "P" | "Q" | "R" | "S" | "T" | "U" |
"V" | "W" | "X" | "Y" | "Z"
маленькая_буква =
"a" | "b" | "c" | "d" | "e" | "f" | "g" |
"h" | "i" | "j" | "k" | "l" | "m" | "n" |
"o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w" | "x" | "y" | "z"
цифра =
"0" | "1" | "2" | "3" | "4" | "5" | "6" |
"7" | "8" | "9"
буквы_и_цифры =
[буквы_и_цифры ["_"]] буква_или_цифра
буква_или_цифра = буква | цифра

К специальным символам относятся:
! " # ’ ( ) ∗ + , − . / : ; < = > ? [\ ] _ ‘ { | }

Глава 2

Лексика

Текст программы рассматривается как последовательность лексем и разделителей. Разделителями являются комментарии, а также пробелы и управляющие символы, не входящие в состав лексем и комментариев. Чтобы обеспечить однозначность трансляции текста, приняты следующие соглашения:

1. Сканирование текста всегда осуществляется слева направо.

2. В состав каждой лексемы включается по возможности большее число графических символов.

3. Фрагмент текста «:–» не является лексемой, если он расположен между лексемами «{» и «}», составляющими пару «открывающая скобка — закрывающая скобка».

4. Фрагмент текста «<–» не является лексемой, если он расположен непосредственно перед числовым литералом или ограничителем «(».

Пример. Последовательность лексем и разделителей.

Текст «P{a:–7}:–P{/∗/b:0}.––1––» содержит лексемы «P», «{», «a», «:», «–», «7», «}», «:–», «P», «{», «b», «:», «0», «}», «.» и комментарии «/∗/», «––1––».

Ссылки: графема 1, комментарий 2.2, лексема 2.1, ограничитель 2.1.5, программа 4, управляющий символ 1, числовой литерал 2.1.3.

2.1 Лексемы

Лексемами являются: переменные, символы и ключевые слова, целые числовые литералы, вещественные числовые литералы, сегменты строк, ограничители.

В ходе сканирования текста происходит преобразование информации, поэтому в определении языка различаются собственно «лексемы», воспринимаемые лексическим анализатором, и «значения лексем», которые обрабатывает синтаксический анализатор.

Ссылки: ключевое слово 2.1.2, ограничитель 2.1.5, переменная 2.1.1, сегмент строки 2.1.4, символ 2.1.2, числовой литерал 2.1.3.

2.1.1 Переменные

Переменная — это имя, начинающееся с большой буквы или символа подчёркивания «_».

переменная =
    большая_буква [["_"] буквы_и_цифры] |
    "_" [буквы_и_цифры]

Маленькие буквы в составе переменной заменяются соответствующими большими буквами, при этом все остальные графемы остаются без изменений. Полученная последовательность графем считается значением лексемы.

Переменная «_» называется «анонимной». Считается, что все анонимные переменные являются некоторыми уникальными, однократно использованными именами.

Пример. Правильно построенные переменные:

A1, _, AbC_Ef_H7, _7, Variable, _X_123

Ссылки: большая буква 1, буквы и цифры 1, графема 1, значение лексемы 2.1, лексема 2.1, маленькая буква 1.

2.1.2 Символы и ключевые слова

Символ — это имя, начинающееся с маленькой буквы или заключённое в апострофы. Различаются простые символы и символы в апострофах:

символ = простой_символ | символ_в_апострофах

простой_символ =
    маленькая_буква [["_"] буквы_и_цифры]
символ_в_апострофах = ' { графема } '

Большие буквы в составе символа заменяются соответствующими маленькими буквами, при этом все остальные графемы остаются без изменений. Полученная последовательность графем считается значением символа. Апострофы, в которые может быть заключён символ, не являются составными частями его значения. Если апострофы не используются, значение символа не должно совпадать с ключевыми словами языка.

Ключевыми словами являются следующие имена:

as	— под_именем
class	— класс
import	— импортировать
from	— из
package	— пакет
project	— проект
protecting	— защищающий
specializing	— специализирующий
suspending	— отключающий

Для написания ключевых слов языка используются только маленькие буквы. Значениями ключевых слов считаются соответствующие цепочки графем.

Пример. Правильно построенные символы:

symbol, 'ALPHA', abc_EF_h, '', s4734

Ссылки: большая буква 1, буквы и цифры 1, графема 1, значение лексемы 2.1, маленькая буква 1, простой символ 2.1.2, символ в апострофах 2.1.2.

2.1.3 Числовые литералы

Числовой литерал — это лексема, обозначающая числовое значение:

числовой_литерал =
    расширенное_число [порядок] |
    цифры "#" расширенное_число "#" [порядок] |
    ` графема

расширенное_число =
буквы_и_цифры ["." буквы_и_цифры]

Числовые литералы бывают целые и вещественные (плавающие) — значениями таких литералов являются, соответственно, (беззнаковые) целые и вещественные числа.

По умолчанию основание числового литерала равно 10. Основание и порядок числовых литералов всегда записываются в десятичной системе. В качестве (расширенных) цифр от 10 до 35 используются латинские буквы от «A» до «Z» (от «a» до «z») соответственно. Значение каждой (расширенной) цифры литерала с основанием должно быть меньше основания.

Числовые литералы, содержащие точку, обозначают вещественные числа. В языке не гарантируется точное представление вещественных чисел, количество значащих цифр которых превышает значение, соответствующее максимальной относительной погрешности D, определяемой конкретной реализацией языка. В качестве значений таких числовых литералов принимаются некоторые близкие числа, отличающиеся от них на величину, не превышающую D.

Если в качестве числового литерала используется последовательность ‘ графема, его значением является числовой код заданного графического символа (целое число) в кодировке, определяемой конкретной реализацией языка. Использование пробела, так же как и управляющих символов в определении числового литерала не допускается (считается синтаксической ошибкой).

цифры = [цифры ["_"]] цифра

Символы подчёркивания между соседними цифрами и буквами числового литерала не влияют на его значение.

порядок = буква_e ["+" | "−"] цифры
буква_e = "E" | "e"

Для получения значения числового литерала с порядком необходимо умножить значение числового литерала без порядка на основание, возведённое в указанную порядком степень. Порядок целых числовых литералов не может содержать знак минус.

Пример. Правильно построенные числовые литералы:
13_274, 2#1100_0100#E4, 39.123e100, 8#177_777#,
3.217_514e+90, 16#EF93#, ‘y, 8#3.51#E−31

Ссылки: буква 1, буквы и цифры 1, графема 1, значение лексемы 2.1, лексема 2.1, управляющий символ 1, цифра 1, числовой литерал 2.1.3.

2.1.4 Сегменты строк

Сегмент строки — это лексема, обозначающая цепочку графических и управляющих символов:

сегмент_строки = ‘"’ { графема | “\” код } ‘"’

В ходе сканирования сегмента строки конструкции вида «\» код (где код — некоторая буква или числовой литерал) заменяются соответствующими графическими и управляющими символами.

код = “b” | “t” | “n” | “v” | “f” | “r” | числовой_литерал

Буквенные коды соответствуют управляющим символам:

b — возврат на одну позицию;	t — горизонтальная табуляция;
n — перевод строки;	v — вертикальная табуляция;
f — перевод формата;	r — возврат каретки.

В качестве кода в сегменте строки не допускается (считается синтаксической ошибкой) использование вещественных числовых литералов, а также числовых литералов, значения которых лежат за пределами некоторого интервала, определяемого конкретной реализацией языка. В случае если графический символ, следующий после «\», не является кодом, переключатель «\» игнорируется, а обнаруженный за ним графический символ включается в сегмент строки без дальнейшего анализа. Полученная таким образом последовательность графических и управляющих символов, не считая кавычек, в которые заключён сегмент строки, является значением сегмента строки.

Пример. Правильно построенные сегменты строк:

"String \"XYZ\"\n", "", "c:\\dos*.*"

Ссылки: буква 1, графема 1, значение лексемы 2.1, лексема 2.1, сегмент строки 2.1.4, управляющий символ 1, числовой литерал 2.1.3.

2.1.5 Ограничители

Ограничитель — это последовательность из одного или нескольких специальных символов, используемая в синтаксических конструкциях языка. В языке используются:

1. простые ограничители
   ! # ( ) ∗ + , − . / : ; < = > ? [ ] { | }
2. составные ограничители
   :− << <− ?? == := <> <= >=

Значениями ограничителей считаются соответствующие цепочки графем.

Ссылки: графема 1, значение лексемы 2.1, специальный символ 1.

2.2 Комментарии

Комментарием является последовательность графических и управляющих символов, начинающаяся с открывающей скобки комментария и заканчивающаяся закрывающей скобкой; комментариям разных типов соответствуют разные скобки. Открывающая скобка не является началом комментария, если её графические символы входят в состав лексемы или другого комментария. Определены два типа комментариев:

1. Однострочный комментарий: открывающая скобка — два соседних дефиса; закрывающая — любой управляющий символ, отличный от горизонтальной табуляции.
2. Многострочный комментарий: открывающая и закрывающая скобки «/∗» и «∗/» соответственно. Повторное вхождение открывающей скобки «/∗» в состав многострочного комментария считается синтаксической ошибкой.

Пример. Правильно построенные комментарии:

−− Однострочный комментарий

/∗/
/∗ Многострочные комментарии ∗/
/∗/

Ссылки: графема 1, лексема 2.1, управляющий символ 1.

Глава 3

Определение данных

В общем случае, термы языка могут обозначать:

1. элементы данных;
2. экземпляры классов;
3. значения лексем «переменная» (если речь идёт о несвязанных переменных).

терм =
    простой_терм |
    составной_терм |
    вызов_функции_в_предложении

Элементы данных создаются в ходе исполнения вызовов предикатов, во время построения слотов миров, а также во время глобальных операций с общими переменными.

В дальнейшем, когда будет идти речь об унификации и других операциях с термами, следует иметь в виду обработку значений термов.

В качестве функторов составных термов и атомарных формул используются символы и метапеременные (метафункторы):

функтор = символ | метапеременная

Метапеременными, называются переменные, используемые в качестве функторов и символов. Метапеременные, используемые в качестве функторов, называются метафункторами.

метапеременная = переменная

В качестве функторов метапеременные разрешается использовать только в составе предложений и только при условии, что такой же метафунктор является именем предиката в заголовке рассматриваемого предложения.

Функтор, используемый в составе определения класса и совпадающий с некоторым атрибутом этого класса, должен быть символом в апострофах.

Ссылки: атом 6.1, атрибут 4.1.1, вызов функции в предложении 6.2.1, глобальные операции 7.2, заголовок предложения 6, значение лексемы 2.1, исполнение предиката 6.3.1, класс 4.1, мир 5.1, несвязанная переменная 3.1, переменная 2.1.1, построение слотов 5.4.2, предложение 6, простой терм 3.1, символ 2.1.2, символ в апострофах 2.1.2, составной терм 3.2, унификация 3.3.

3.1 Простые термы

Простой терм — это элементарная синтаксическая конструкция, обозначающая данные и миры. Простыми термами являются константы (символ, целое число, вещественное число, строковый литерал, спейсер #, метапеременная, обозначающая терм в метапредложении), а также параметры:

простой_терм = константа | параметр
константа =
    символ_в_апострофах |
    ["–"] числовой_литерал |
    строковый_литерал |
    "#" |
    метапеременная

Число обозначается с помощью числового литерала, перед которым может стоять знак минус. В целых и вещественных числах с явно указанным основанием использовать знак минус не разрешается.

Строковый литерал — это последовательность сегментов строки, обозначающая цепочку графических и управляющих символов:

строковый_литерал = [строковый_литерал] сегмент_строки

Спейсер # обозначает неизвестный элемент данных или мир.

Метапеременные разрешается использовать в качестве простых термов только в составе предложений и только при условии, что такая же метапеременная является именем предиката или атомарной формулой в заголовке рассматриваемого предложения.

параметр = переменная | атрибут

Считается, что значением терма «переменная» является значение лексемы «переменная», до тех пор пока эта переменная (терм) не будет связана с какой-либо константой, составным термом или миром. Значением связанной переменной считается соответствующий элемент данных, мир или спейсер. Переменная, не связанная с константой, составным термом или миром, называется «несвязанной».

Значения других простых термов определяются значениями соответствующих им лексем.

Диапазоны допустимых целых и вещественных чисел определяются конкретной реализацией языка. При этом значения числовых литералов с явно указанным основанием (выходящие за пределы допустимого диапазона) разрешается использовать в качестве битового представления отрицательных чисел.

Значением строкового литерала является конкатенация значений последовательности входящих в его состав сегментов строк. Максимальная допустимая длина значения строкового литерала определяется конкретной реализацией языка.

Пример. Правильно построенные простые термы:

VARIABLE, 'symbol', 2#0100_1100#, −34.0e−9, "A" "TEXT" "LINE\n"

Ссылки: атом 6.1, атрибут 4.1.1, графема 1, данные 3, заголовок предложения 6, значение лексемы 2.1, значение терма 3, лексема 2.1, метапеременная 3, метапредложение 6, мир 5.1, переменная 2.1.1, предложение 6, простой терм 3.1, связывание 3.3, сегмент строки 2.1.4, символ 2.1.2, символ в апострофах 2.1.2, составной терм 3.2, строковый литерал 3.1, терм 3, управляющий символ 1, числовой литерал 2.1.3.

3.2 Составные термы

Составными термами являются структуры, списки и недоопределённые множества:

составной_терм =
структура | список | недоопределённое_множество

Значения составных термов определяются с помощью кортежей и некоторых специальных имён (констант).

Ссылки: значение терма 3, недоопределённое множество 3.2.3, составной терм 3.2, список 3.2.2, структура 3.2.1, терм 3.

3.2.1 Структуры

Структура — это составной терм, построенный из функтора и последовательности одного или более аргументов, заключённой в круглые скобки:

структура = функтор "(" термы_и_выражения ")"
термы_и_выражения =
[термы_и_выражения ","] терм_или_выражение
терм_или_выражение = терм | выражение

Значением структуры $f(A_1,A_2,\ldots,A_n)$ является кортеж длины $n+2$, в первой позиции которого стоит специальная константа structure:

$$<\text{structure},f,A_1,A_2,\ldots,A_n>.$$

Пример. Правильно построенные структуры:
g1(1+2,X,Y), functor(i(1−(R∗12),2,3),4,k(5),Z), h(J)

Ссылки: выражение 6.2.2, значение терма 3, составной терм 3.2, терм 3, функтор 3.

3.2.2 Списки

Список — это составной терм, построенный из последовательности (возможно, пустой) аргументов, заключённой в квадратные скобки. В случае если последовательность аргументов списка не является пустой, в его состав может быть включён дополнительный компонент, обозначающий остаток (хвост) списка:

список = "[" [термы_и_выражения ["|" хвост]] "]"
хвост = параметр | вызов_функции_в_предложении | выражение

Значением пустого списка [ ] является специальная константа

#empty_list.

Значением списка [A_1,A_2,...,A_n|Rest] является кортеж

<list,A_1,<list,A_2,...<list,A_n,Rest>...>>,

где list — специальная константа, Rest — хвост списка.

Таким образом, терму [A_1,A_2,...,A_n] соответствует значение

<list,A_1,<list,A_2,...<list,A_n,#empty_list>...>>.

Пример. Правильно построенные списки:
[17,_,"item_of_list",321,93,_], [X+721,Y,R+H,Z|R], [ ]

Ссылки: вызов функции в предложении 6.2.1, выражение 6.2.2, значение терма 3, параметр 3.1, составной терм 3.2, терм 3, термы и выражения 3.2.1.

3.2.3 Недоопределённые множества

Недоопределённое множество — это составной терм, построенный из набора (возможно, пустого) элементов, заключённого в фигурные скобки. Элементы недоопределённого множества задаются в виде пар

«имя_элемента: значение_элемента»,

где имя элемента — некоторый символ или неотрицательное целое число, а значение элемента — терм или выражение:

элементы_множества =
[элементы_множества ","] элемент_множества
элемент_множества =
имя_элемента [":" терм_или_выражение] | атрибут
имя_элемента = символ | числовой_литерал

Если в составе элемента множества не заданы терм или выражение после имени элемента, значением такого элемента считается анонимная переменная «_». В таких случаях символы, используемые в качестве имён элементов, обязательно должны быть в апострофах.

Если недоопределённое множество используется в составе определения класса, то имена элементов множества, совпадающие с атрибутами этого класса, должны быть символами в апострофах.

Если в качестве элемента множества задан атрибут Name, то именем элемента множества считается символ 'Name', а значением — слот Name.

Заголовком недоопределённого множества называется значение элемента с именем 0 (ноль), которое может быть задано в начале недоопределённого множества, за пределами фигурных скобок. При таком способе определения в качестве заголовка недоопределённого множества разрешается использовать только простые термы:
Недоопределённое множество вида

$$F\{x_1{:}A_1,x_2{:}A_2,\ldots,x_n{:}A_n|Rest\},$$

в составе которого задан заголовок F, эквивалентно

$$\{0{:}F,x_1{:}A_1,x_2{:}A_2,\ldots,x_n{:}A_n|Rest\}.$$

В случае если набор элементов множества (учитывая заголовок) не является пустым, в составе множества может быть задан дополнительный компонент, обозначающий неопределённый остаток (хвост) множества. Если недоопределённое множество используется в составе определения атрибутов класса, в качестве хвоста этого множества разрешается использовать только переменные.

недоопределённое_множество =
[простой_терм] "{" элементы_и_хвост_множества "}"
элементы_и_хвост_множества = [элементы_множеств] ["|" хвост]

Недоопределённое множество не может содержать пары с одинаковыми именами элементов.

Для того чтобы построить значение недоопределённого множества, необходимо:

1. Просмотреть полный текст программы и построить множество S всех имён элементов всех недоопределённых множеств, которые в ней используются.

2. С помощью лексикографического упорядочения из элементов множества S построить цепочку $s_1,s_2,\ldots,s_m$ (m — мощность множества S).

3. Если недоопределённое множество обозначает конечное число элементов (случай $\{s_x{:}A_x,s_y{:}A_y,\ldots,s_z{:}A_z\}$), его значением является кортеж длины $m{+}1$:

$<$set,. . . ,z,. . . ,t(A_y),. . . ,z,. . . ,t(A_x),. . . ,t(A_z),. . . $>$,

где set — специальная константа, t — имя вспомогательного функтора. Каждая позиция $i{+}1$ ($i{=}1,\ldots,m$) кортежа содержит значение t(A_i), если пара с именем элемента s_i присутствует в рассматриваемом терме, или, если такая пара не обнаружена, специальную константу z. Значением пустого множества {}, в частности, является кортеж вида

$<$set,z,z,z,z,. . . z,z,z$>$.

4. Значением недоопределённого множества общего вида

{s_x:A_x,s_y:A_y,. . . ,s_z:A_z|Rest}

является кортеж длины $m{+}1$:

G_1 = $<$set,. . . ,V_1,. . . ,t(A_y),. . . ,V_2,. . . ,t(A_x),. . . $>$,

каждая позиция $i{+}1$ ($i{=}1,\ldots,m$) которого содержит t(A_i), если пара с именем элемента s_i присутствует в рассматриваемом терме, или некоторую уникальную переменную V_j, если соответствующая пара не обнаружена. Кроме того, понадобится ещё один кортеж

G_2 = $<$set,. . . ,V_1,. . . ,z,. . . ,V_2,. . . ,z,. . . $>$,

который отличается от предыдущего тем, что все аргументы t(A) в нём заменены константами z. Считается, что всякий раз, когда создаётся значение недоопределённого множества G_1, одновременно с этим происходит унификация G_2 с переменной Rest.

Пример. Правильно построенные недоопределённые множества:
R2{x:17,y:−(1+X),'z',5:'yes'}, {q:Y,e:W,symbol:_,k:0|W}, {}

Ссылки: анонимная переменная 2.1.1, атрибут 4.1.1, выражение 6.2.2, значение терма 3, имя элемента 3.2.3, класс 4.1, недоопределённое множество 3.2.3, переменная 2.1.1, программа 4, простой терм 3.1, символ 2.1.2, символ в апострофах 2.1.2, слот 5.1, составной терм 3.2, терм 3, терм или выражение 3.2.1, унификация 3.3, функтор 3, хвост 3.2.2, число 3.1, числовой литерал 2.1.3, элемент множества 3.2.3.

3.3 Унификация термов

Унификацией называется операция сравнения (отождествления) нескольких формул, связывающая переменные в составе формул сопоставленными с ними подформулами.

Унификация термов осуществляется в ходе исполнения вызовов предикатов, в момент создания значений недоопределённых множеств, а также во время глобальных операций с общими переменными. Кроме того, унификация термов может быть вызвана явно с помощью встроенного предиката «унифицировать термы»:

L == R.

Встроенный предикат '==' разрешается использовать с произвольным количеством аргументов:

'=='(V_1,...,V_k).

Унификация несвязанной переменной с константой, составным термом или миром вызывает «связывание» этой переменной — замену всех вхождений этой переменной соответствующим элементом данных или миром.

Унификация различных несвязанных переменных вызывает их «сцепление» (отождествление): в дальнейшем любое связывание одной из сцеплённых переменных автоматически вызывает такое же связывание всех сцеплённых с ней переменных.

В Акторном Прологе область действия операций связывания и сцепления переменной всегда ограничена множеством её вхождений, принадлежащих некоторым конкретным акторам.

В Акторном Прологе допускается унификация целых и вещественных чисел. Унификация целого и вещественного чисел заканчивается успехом тогда и только тогда, когда вещественное число обозначает целую величину, равную унифицируемому с ней целому числу.

Константы и составные термы несопоставимы между собой (их унификация невозможна). Невозможна также унификация констант и составных термов с экземплярами классов. Унификация термов, обозначающих миры, возможна лишь в том случае, если они обозначают один и тот же мир. Спейсер не может быть унифицирован ни с какой другой константой, кроме себя.

Проверкой вхождения называется специальная операция, осуществляемая в ходе унификации, предотвращающая (запрещающая) связывание

переменной с составными термами, содержащими эту переменную. В соответствии с семантикой Акторного Пролога, проверка вхождения не распространяется на переменные в составе миров, являющихся компонентами унифицируемых термов.

В максимальной версии языка в ходе унификации осуществляется проверка вхождения. В быстрой версии проверка вхождения используется только во время глобальных операций с общими переменными. В минимальной версии языка проверка вхождения не используется, однако во время глобальных операций, при обнаружении переменной, связанной с термом, содержащим эту переменную, допускается вызов исключительных ситуаций.

Пример. Унификация двух составных термов:

{region:X,name:"Baikal"|Rest1}
==
{name:Y,object:'lake',region:"Siberia"}

В ходе унификации элементы одного множества будут унифицированы с элементами другого в соответствии с заданными именами элементов. Результатом унификации станут подстановки X="Siberia", Y="Baikal", на месте переменной Rest1 окажется значение G недоопределённого множества, включающего одну-единственную пару object:'lake'. Остальные позиции кортежа G (кроме первой, которая всегда содержит set) будут заполнены z.

Ссылки: актор 7.1, встроенный предикат 8, глобальные операции 7.2, данные 3, значение терма 3, имя элемента 3.2.3, исключительная ситуация 7.5, исполнение предиката 6.3.1, константа 3.1, мир 5.1, недоопределённое множество 3.2.3, несвязанная переменная 3.1, переменная 2.1.1, принадлежать актору 7.2, составной терм 3.2, спейсер 3.1, терм 3, число 3.1, элемент множества 3.2.3.

Глава 4

Структура программы

Программа состоит из множества классов и целевого утверждения («проекта»):

программа = { определение_класса | определение_проекта }

Будем говорить, что некоторый класс C (или проект) «использует» класс E, если E является предком C в иерархии наследования классов, а также если C (проект) или кто-либо из его предков содержит конструктор экземпляра класса E (или конструктор экземпляра класса F, такого что класс F использует класс E), не считая тех конструкторов, которые входят в состав инициализаторов, перекрываемых во время построения соответствующих миров.

В программе должны быть определены все классы, используемые проектом.

Исполнением программы называется построение и дальнейшее согласование некоторых процессов. Исполнение программы начинается с доказательства конструктора процесса, заданного в определении проекта, а также формирования процесса, построенного в результате доказательства этого конструктора.

Ссылки: иерархия наследования 4.1, инициализатор 4.1.2, класс 4.1, конструктор 4.1.3, конструктор процесса 4.1.3, мир 5.1, перекрытие инициализаторов 5.4.2, построение миров 5.4.1, построение процесса 5.4.1, проект 4.2, процесс 5.2, согласование процессов 7.4, формирование процесса 5.4.1.

4.1 Классы

Класс — это набор предложений языка, имеющий уникальное имя и входящий в состав иерархии наследования:

определение_класса =
class заголовок_класса ":" атрибуты "[" предложения "]"

В языке используется одиночное наследование: у класса может быть не более одного непосредственного предка и неограниченное число потомков. Имя непосредственного предка указывается в определении после имени класса:

заголовок_класса =
имя_класса [**specializing** имя_класса]
имя_класса = символ_в_апострофах

В иерархии наследования классов, используемых проектом, запрещены циклические зависимости.

Примечание. Неаккуратное (взаимно-) рекурсивное использование классов может приводить к бесконечному увеличению количества миров в ходе формирования экземпляров классов.

Пример. Правильно построенный класс.

```
class 'MyWindow' specializing 'Report':
text_color = 'Green';
[
goal:-
    show,!.
]
```

Ссылки: атрибуты 4.1.1, имя класса 4.1, использование класса 4, мир 5.1, отсечение 8, предложение 6, проект 4.2, символ в апострофах 2.1.2, формирование миров 5.4.1, **class** 2.1.2, goal 5.4.1, **specializing** 2.1.2.

4.1.1 Атрибуты классов

Атрибутами называются имена слотов экземпляра класса, определяемые в составе класса. Каждый атрибут должен быть объявлен во всех классах, связанных отношением наследования, в которых используется

соответствующий слот. Область действия атрибута распространяется на инициализаторы слотов в определении атрибутов класса, а также на все предложения класса.

атрибуты = { определение_атрибута ";" }
определение_атрибута =
[описатель_порта ":"] атрибут ["=" инициализатор]
описатель_порта = **suspending** | **protecting**
атрибут = простой_символ

В составе инициализаторов слотов могут использоваться переменные. Область действия таких переменных ограничена множеством инициализаторов слотов в определении атрибутов класса. В определении атрибутов класса не допускается однократное использование переменных, отличных от «_».

Атрибут self — предопределённый, он обозначает непосредственно тот экземпляр класса, в котором это имя используется.

Повторное определение атрибутов класса (в том числе переопределение атрибута self) считается синтаксической ошибкой.

Пример. Правильно определённые атрибуты класса:

```
a = Y;                          e = ('Q',x='+'(a,f),m=self,k=e);
b;                              f = '*'(Y,7);
c = 'f'(_,[3,7],Y,a);           d = [];
g = _;                          h = {x:1,y:Y,z:R|_};
i = [0,_,j|R];                  j = a;
```

Ссылки: инициализатор 4.1.2, класс 4.1, мир 5.1, переменная 2.1.1, предложение 6, простой символ 2.1.2, слот 5.1, **protecting** 2.1.2, **suspending** 2.1.2.

4.1.2 Инициализаторы слотов

Инициализатором слота называется синтаксическая конструкция, определяющая начальное значение слота:

инициализатор = терм | конструктор

Ссылки: конструктор 4.1.3, начальное значение слота 5.4.2, слот 5.1, терм 3.

4.1.3 Конструкторы

Конструктором называется утверждение о существовании экземпляра класса или резидента. В результате доказательства конструкторов происходит построение новых экземпляров классов и резидентов.

Различаются конструкторы миров (а именно простые конструкторы и конструкторы процессов), а также конструкторы резидентов.

конструктор = конструктор_мира | конструктор_резидента
конструктор_мира =
простой_конструктор | конструктор_процесса

Простым конструктором называется элементарное логическое утверждение о существовании экземпляра класса.

простой_конструктор =
"(" имя_класса { "," определение_атрибута } ")"

Конструктор процесса — это утверждение о существовании процесса. Доказательство конструктора процесса приводит к созданию нового процесса.

конструктор_процесса = "(" простой_конструктор ")"

Аргументы конструктора мира определяют значения слотов соответствующего экземпляра класса (значения слотов процесса).

Отсутствие инициализатора в определении некоторого атрибута конструктора с именем Name является допустимым только в том случае, если рассматриваемый конструктор экземпляра класса находится в области действия слота с именем Name. Такое определение атрибута эквивалентно определению вида «Name=Name».

В конструкторе экземпляра класса не допускается определение нескольких атрибутов с одинаковыми именами. Не допускается также определение атрибута self.

Конструктором резидента называется синтаксическая конструкция, определяющая резидента. Конструктор резидента определяет целевые миры резидента и соответствующую резиденту атомарную формулу. Доказательство конструктора резидента приводит к созданию нового резидента.

конструктор_резидента =
[параметр_или_конструктор] "??" простой_атом

параметр_или_конструктор =
целевой_параметр | конструктор_мира
целевой_параметр = параметр

Атомарная формула в составе конструктора резидента является вызовом функции. В качестве простых атомов в конструкторе резидента не разрешается использовать метаатомы.

Пример. Правильно построенные конструкторы:
('R53',a=1,b=7,c=_,'d'), (('W',q=3)), ('E') ?? f(A)

Ссылки: атом 6.1, атрибут 4.1.1, вызов функции 6.2.1, значение слота 5.1, имя класса 4.1, инициализатор 4.1.2, конструктор мира 4.1.3, конструктор процесса 4.1.3, конструктор резидента 4.1.3, метаатом 6.1.1, мир 5.1, параметр 3.1, построение миров 5.4.1, простой атом 6.1.1, простой конструктор 4.1.3, процесс 5.2, резидент 5.3, слот 5.1, целевой мир 5.3, self 4.1.1.

4.2 Проект

Проектом называется целевое утверждение программы — некоторый конструктор процесса:

определение_проекта =
project ":" конструктор_процесса

В составе проекта могут использоваться переменные. Область действия таких переменных ограничена пределами проекта. В определении проекта не допускается однократное использование переменных, отличных от «_».

В качестве обозначения экземпляра класса, соответствующего проекту, в определении проекта допускается использование предопределённого атрибута self.

Проект доказывается по правилам исполнения конструкторов процессов.

Пример. Правильно построенный проект.

project: (('P',x=[1,7,9|W],y=W,p=self))

Ссылки: атрибут 4.1.1, исполнение конструктора 5.4.1, конструктор процесса 4.1.3, мир 5.1, переменная 2.1.1, программа 4, **project** 2.1.2, self 4.1.1.

4.3 Пакеты

Пакетом называется совокупность классов, связанных между собой по смыслу. Текст программы может состоять из нескольких пакетов. Каждый пакет должен храниться в отдельном исходном файле. Каждый исходный файл программы считается отдельным пакетом.

Каждый пакет обладает собственной областью видимости имён классов. Таким образом, имена классов, используемые внутри пакета, не видны из других пакетов до тех пор, пока не будут импортированы в эти пакеты явным образом, с помощью команд импорта. Целевое утверждение (проект) программы может входить в состав любого пакета.

В общем случае, каждый пакет включает заголовок пакета, команды импорта, а также произвольное количество классов и (возможно) целевое утверждение (проект):

пакет = [заголовок_пакета] команды_импорта программа
заголовок_пакета = **package** имя_пакета ":"
имя_пакета = строковый_литерал

В начале исходного файла может быть указан заголовок пакета, определяющий имя пакета. Именем пакета является строковый литерал, представляющий собой имя соответствующего исходного файла без расширения. В качестве имени пакета допускается использование:

1. Полного имени файла, включающего все подкаталоги.

2. Укороченного имени файла, если пакет расположен в одном из подкаталогов системного каталога, который должен быть определён в конкретной реализации языка. В этом случае имя пакета должно включать лишь подкаталоги, вложенные по отношению к системному каталогу.

В качестве разделителя имён подкаталогов в имени пакета допускается использование как прямой «/», так и обратной «\\» дробной черты (реализация языка должна одинаково интерпретировать оба варианта). Имя пакета должно соответствовать реальному имени исходного файла. Несоответствие имени транслируемого пакета, не позволяющее однозначно сопоставить его с именем исходного файла, является синтаксической ошибкой. Если заголовок пакета не задан, имя пакета определяется автоматически по имени соответствующего исходного файла.

Команда импорта делает видимым в пакете имя класса (заданное после ключевого слова import), определённого в некотором другом пакете (имя которого указано после ключевого слова from). Класс, заданный в команде импорта, называется импортируемым классом. Действия, осуществляемые командой импорта называются импортом класса.

команды_импорта = { команда_импорта }
команда_импорта =
import импортируемое_имя **from** имя_пакета ";"
импортируемое_имя = имя_класса [**as** имя_класса]

Если в составе команды импорта задано ключевое слово as, импортируемый класс будет видим в текущем пакете под именем, заданным после ключевого слова as. Если ключевое слово as не задано, импортируемый класс будет видим под своим собственным именем.

Последовательность команд импорта, которая делает видимыми некоторые импортируемые классы под одним и тем же именем, является синтаксической ошибкой. Использование в пакете нескольких команд импорта, которые делают видимым один и тот же класс из одного и того же пакета, является синтаксической ошибкой.

Пример. Правильно построенные команды импорта:

```
import 'Sphere' as 'Marker' from "VRML/Shapes";
import 'Analyzer' from "My\\Features";
```

Ссылки: заголовок пакета 4.3, имя класса 4.1, имя пакета 4.3, исходный файл 4.4, класс 4.1, ключевое слово 2.1.2, команда импорта 4.3, программа 4, проект 4.2, строковый литерал 3.1, **as** 2.1.2, **from** 2.1.2, **import** 2.1.2, **package** 2.1.2.

4.4 Трансляция исходных файлов

Исходные файлы программы могут транслироваться отдельно друг от друга или совместно (конкретная реализация языка может поддерживать один из указанных способов или оба способа трансляции). В конкретной реализации языка может осуществляться автоматическая трансляция пакетов, указанных в командах импорта в составе других пакетов. В зависимости

от реализации языка, автоматическая трансляция пакетов может рассматриваться как трансляция этих пакетов отдельно от других пакетов или как совместная трансляция этих пакетов.

Определения классов и определение проекта являются «элементарными программными модулями», из которых строится исходный файл. Результат их трансляции — добавление или замена соответствующих «библиотечных модулей» в некоторой программной библиотеке, структура которой должна быть определена в конкретной реализации языка. Набор совместно транслируемых пакетов, так же как каждый отдельный пакет не могут содержать повторные определения классов. Допускается только одно определение проекта в каждом отдельном пакете и в наборе совместно транслируемых пакетов. Максимальная допустимая длина исходного файла определяется конкретной реализацией языка.

Формированием программы называется сборка программы из библиотечных модулей, сопровождаемая проверкой её синтаксической правильности. Формирование программы может осуществляться перед началом или во время её исполнения, однако синтаксически правильной считается только такая программа, которая может быть сформирована перед началом исполнения.

Примечание. Некоторые синтаксические ошибки обнаруживаются лишь на этапе формирования программы из библиотечных модулей. Такими ошибками являются отсутствие определения проекта или некоторых определений классов в программной библиотеке, а также циклы в иерархии наследования.

Пример. Исходный файл программы.

```
package "Examples/Example1":

class 'Hello'                          -- Определение класса
specializing 'Report':
width     = 25;                        -- Атрибуты
height    = 10;
[                                      -- Предложения
goal:-
    write("Hello world !").
]
```

```
project:                                    -- Определение проекта
    (('Hello', x=1, y=2, width=30, height=7))
```

Ссылки: атрибут 4.1.1, иерархия наследования 4.1, исполнение программы 4, класс 4.1, команда импорта 4.3, пакет 4.3, предложение 6, программа 4, проект 4.2, **class** 2.1.2, goal 5.4.1, **package** 2.1.2, **project** 2.1.2, **specializing** 2.1.2.

Глава 5

Структура пространства поиска

Составными частями пространства поиска служат экземпляры классов. В общем случае, пространство поиска развёртывается динамически, в ходе исполнения программы.

Ссылки: исполнение программы 4, мир 5.1.

5.1 Экземпляры классов

Экземпляр класса («мир») — это конкретное применение класса. В состав экземпляра класса входят:

1. Предложения класса, а также предложения его предков.
2. Слоты экземпляра класса.

Слот — это составная часть экземпляра класса, характеризуемая именем и значением. Именем слота является некоторый атрибут, а значением слота — терм.

Построение экземпляров классов происходит в результате доказательства утверждений об их существовании — конструкторов.

Мир B называется вложенным по отношению к миру A, если конструктор мира B является инициализатором некоторого слота мира A или какого-либо мира E, вложенного по отношению к A.

Пример. Наследование предложений класса.

Экземпляр класса 'CHERRY' содержит предложения colour и taste, определённые в классах 'CHERRY' и 'FRUIT'.

```
class 'FRUIT':
[
taste('sweet').
taste('sour').
]
class 'CHERRY' specializing 'FRUIT':
[
colour('red').
]
```

Ссылки: атрибут 4.1.1, вложенность миров 5, инициализатор 4.1.2, класс 4.1, конструктор 4.1.3, конструктор мира 4.1.3, построение миров 5.4.1, предложение 6, терм 3, **class** 2.1.2, **specializing** 2.1.2.

5.2 Процессы

Процессом называется экземпляр класса, предложения которого исполняются параллельно по отношению к предложениям других процессов. Процессам соответствуют отдельные части пространства поиска, не пересекающиеся с другими процессами.

Построение процессов осуществляется в результате доказательства конструкторов процессов. Создателем процесса называется процесс, одному из слотов миров которого соответствовал инициализатор — конструктор рассматриваемого процесса.

Считается, что некоторый актор «принадлежит» процессу G, если этот актор доказывается, доказан или должен быть доказан в мире, входящем в состав процесса G. Исполнением процесса называется доказательство акторов, принадлежащих этому процессу.

«Фазами» исполнения процесса называются законченные периоды исполнения процесса, соответствующие:

- обработке процессом сообщений;
- изменению состояния процесса.

После (успешного) окончания очередной фазы исполнения процесса осуществляется «фиксирование» процесса, а именно:

1. Устраняются все точки выбора, возникшие в течение этой фазы.
2. Фиксируются все общие переменные всех акторов, принадлежащих процессу.

Процесс, находящийся на очередной фазе своего исполнения, называется активным.

В быстрой и максимальной версиях языка гарантируется, что исполнение любого процесса программы не может привести к неопределённо длительной приостановке других процессов. В минимальной версии допускается поочерёдное исполнение различных процессов.

Считается, что некоторые процессы «согласованы» между собой, если:

1. Все они находятся в состояниях «доказан» и «неиспользуемый».
2. Не требуется обработка потоковых и прямых сообщений процессами, находящимися в состоянии «доказан».
3. Не требуется обработка потоковых сообщений процессами, находящимися в состоянии «неиспользуемый».
4. Производные значения общих переменных всех процессов могут быть унифицированы.

Примечание. В языке используется только асинхронное взаимодействие между процессами, поэтому предикаты каждого процесса обладают декларативной семантикой, не зависящей от других процессов.

Ссылки: актор 7.1, доказанный процесс 5.2.1, доказательство актора 6.3.1, инициализатор 4.1.2, исполнение предложения 6.3.2, конструктор процесса 4.1.3, мир 5.1, неиспользуемый процесс 5.2.1, обработка сообщения 7.4, общие переменные 7.2, построение процесса 5.4.1, потоковые сообщения 7.4.3, предложение 6, программа 4, производные значения 7.2, прямые сообщения 7.4.2, слот 5.1, сообщение 7.4, состояние процесса 5.2.1, унификация 3.3, фиксирование терма 7.2.

5.2.1 Состояния процесса

В каждый конкретный момент времени процесс находится в одном из трёх состояний:

1. «объявленный»;

2. «используемый»;

3. «неиспользуемый».

В состоянии «объявленный» процесс находится сразу после его создания. Объявленный процесс характеризуется тем, что соответствующие ему экземпляры классов ещё не сформированы. Пока процесс находится в состоянии «объявленный», обработка любых сообщений этим процессом откладывается. Считается, что объявленный процесс не имеет никаких производных значений, и что его акторы несогласованы. После формирования процесса он переходит в состояние «сформированный».

«Используемый процесс» — это обобщающее название для следующих трёх состояний процесса:

1. «сформированный»;

2. «доказанный»;

3. «неудачный».

Состояние «неиспользуемый» характеризуется тем, что на некоторые отключающие порты процесса поданы задерживающие значения. Пока процесс находится в состоянии «неиспользуемый», обработка любых сообщений этим процессом откладывается. Считается, что неиспользуемый процесс не имеет никаких производных значений, и все его акторы согласованы.

Переход процесса в состояние «неиспользуемый» называется «отключением» процесса. Переход процесса в состояние «используемый» называется «подключением» процесса.

Переключение между состояниями процесса «используемый» и «неиспользуемый» происходит автоматически при получении им определённых разновидностей потоковых сообщений. При переходе из состояния «неиспользуемый» в состояние «используемый», процесс всегда оказывается в том конкретном состоянии, в котором он находился до перехода в состояние «неиспользуемый». Если до перехода в состояние «неиспользуемый» процесс находился в состоянии «объявленный», он автоматически переводится в состояние «сформированный».

Переключение между различными разновидностями состояния «используемый» происходит в зависимости от результатов очередной фазы исполнения соответствующего процесса:

Состояние процесса «сформированный» характеризуется тем, что некоторые акторы процесса ещё ни разу не были доказаны и, следовательно, не согласованы. В состоянии «сформированный» процесс может обрабатывать переключающие сообщения, однако обработка любых информационных сообщений откладывается. Фаза исполнения процесса, перед началом которой он находился в состоянии «сформированный», называется инициализацией процесса. После завершения фазы инициализации процесс может перейти в состояние «доказанный» или остаться в состоянии «сформированный».

Состояние «доказанный» («доказан») характеризуется тем, что все акторы, принадлежащие процессу, согласованы. В этом состоянии процесс может обрабатывать как переключающие, так и информационные сообщения. После завершения фазы обработки сообщения процесс переходит в состояние «доказанный» или «неудачный».

Состояние «неудачный» характеризуется тем, что акторы процесса выведены из согласованного состояния. В этом состоянии процесс может обрабатывать переключающие сообщения, а обработка информационных сообщений откладывается. После завершения очередной фазы обработки сообщения процесс может перейти в состояние «доказанный» или остаться в состоянии «неудачный». Переход процесса в состояние «неудачный» называется «нейтрализацией» процесса.

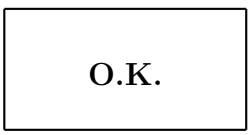
Примечание. Для различных состояний процесса рекомендуются следующие графические обозначения:

объявленный процесс

используемый процесс

неиспользуемый процесс

сформированный процесс



доказанный процесс

неудачный процесс

Ссылки: актор 7.1, доказательство актора 6.3.1, задерживающие значения 6.3.2, информационные сообщения 7.4.1, исполнение процесса 5.2, мир 5.1, обработка сообщения 7.4, отключающий порт 5.2.2, переключающие сообщения 7.4.1, построение процесса 5.4.1, потоковые сообщения 7.4.3, принадлежать процессу 5.2, производные значения 7.2, процесс 5.2, согласование акторов 7.3, согласованность акторов 7.2, сообщение 7.4, фаза 5.2, формирование миров 5.4.1, формирование процесса 5.4.1.

5.2.2 Порты процессов

Переменные процесса G, которые могут принадлежать акторам других процессов, называются портами процесса G.

В ходе исполнения программы каждому порту процесса ставятся в соответствие:

1. Сорт порта: простой, отключающий или защищающий.
2. Состояние порта — вспомогательное логическое значение: «согласованный» или «несогласованный».
3. Актор-представитель порта процесса — некоторый вспомогательный актор, принадлежащий процессу.
4. Текущее значение порта — некоторый вспомогательный терм.
5. Сорт текущего значения порта — вспомогательное логическое значение: «защищённое» или «незащищённое».
6. Производитель текущего значения порта — процесс, построивший текущее значение порта.

Акторы-представители портов процесса предназначены для хранения информации, приходящей в процесс через порты в виде потоковых сообщений.

В ходе обработки потокового сообщения, пришедшего в процесс через некоторый порт S, в начале соответствующей фазы исполнения процесса, осуществляется «активизация» порта S: (единственное) локальное значение актора-представителя порта S устанавливается равным значению порта S на момент начала рассматриваемой фазы исполнения процесса, после чего актор-представитель этого порта объявляется активным (считается, что доказательство этого актора успешно завершено).

Акторы-представители, не активизированные в начале фазы исполнения процесса, используются в ходе исполнения процесса наравне с другими акторами процесса. При этом, однако, в случае нейтрализации акторов-представителей, повторное доказательство этих акторов не осуществляется, и они остаются нейтральными до очередной активизации соответствующих портов.

Порты процесса создаются (определяются) в ходе формирования процесса.

Процесс относит каждый из своих портов к одному из трёх сортов:

1. простой;
2. отключающий;
3. защищающий.

Отключающими портами процессов называется разновидность портов, обладающая следующими свойствами:

1. При получении через отключающий порт задерживающего значения процесс автоматически переводится в состояние «неиспользуемый». Когда значения всех отключающих портов процесса перестают быть задерживающими, он автоматически возвращается в состояние «используемый» (см. правила перехода процесса из состояния «неиспользуемый» в состояние «используемый» в разделе 5.2.1).

2. Отключающий порт R процесса G всегда активизируется в начале фазы исполнения процесса G, если производителем текущего значения порта R является процесс, отличный от G.

Защищающими портами процессов называется разновидность портов, обладающая следующими свойствами:

1. Все потоковые сообщения, передаваемые процессом через защищающий порт автоматически объявляются защищёнными.

2. Значения всех незащищённых сообщений, принимаемых процессом через защищающий порт игнорируются в ходе обработки этих сообщений (активизация порта не осуществляется).

Если порт не является отключающим и не является защищающим, он называется (является) простым.

Сорта портов задаются с помощью описателей портов или по умолчанию. Описателями портов служат ключевые слова «suspending» и «protecting», обозначающие «отключающий» и «защищающий» соответственно.

Если некоторому порту процесса не поставлено в соответствие никаких описателей, этот порт является простым. Если в тексте программы некоторому порту процесса поставлены в соответствие оба описателя «suspending» и «protecting», порт является отключающим.

Примечание. Локальные значения акторов-представителей используются при исполнении встроенного оператора copy вместе с локальными значениями других акторов процесса (см. раздел 8.2).

Примечание. Для портов различных сортов рекомендуются следующие графические обозначения:

Ссылки: активизация актора 7.1, активные акторы 7.1, актор 7.1, встроенный оператор 8, доказательство актора 6.3.1, задерживающие значения 6.3.2, защищённое сообщение 7.4.3, значение потокового сообщения 7.4.3, исполнение предиката 6.3.1, исполнение программы 4, исполнение процесса 5.2, используемый процесс 5.2.1, ключевое слово 2.1.2, локальные значения 7.2, незащищённое сообщение 7.4.3, неиспользуемый процесс 5.2.1, нейтрализация актора 7.1, нейтральные акторы 7.1, обработка потокового сообщения 7.4.3, описатель порта 4.1.1, передача потокового сообщения 7.4.3, переменная 2.1.1, повторные доказательства 7.1, потоковые сообщения 7.4.3, принадлежать актору 7.2, принадлежать процессу 5.2, программа 4, процесс 5.2, сообщение 7.4, состояние процесса 5.2.1, терм 3, фаза 5.2, формирование процесса 5.4.1, copy 8.2, **protecting** 2.1.2, **suspending** 2.1.2.

5.3 Резиденты

Резидентом называется специальная активная сущность, отслеживающая состояния некоторых (целевых) процессов и передающая собранную информацию своему владельцу. Резиденты создаются в результате доказательства конструкторов резидентов.

В общем случае, каждому резиденту соответствуют:

1. Процесс, являющийся «владельцем» резидента.
2. Атомарная формула (вызов функции), заданная в конструкторе резидента.
3. Некоторые «целевые» миры резидента. Процессы, в состав которых входят целевые миры резидента, называются «целевыми» процессами резидента.
4. Некоторые переменные, являющиеся общими для резидента и его владельца.

Владельцем (создателем) резидента является процесс, одному из слотов миров которого соответствовал инициализатор — конструктор рассматриваемого резидента. Резидент взаимодействует со своим владельцем как некоторый процесс — с помощью переключающих потоковых сообщений, по правилам передачи потоковых сообщений, указанным в разделе 7.4.3.

Целевыми мирами резидента считаются все экземпляры классов, входящие в состав значения целевого параметра в текущий момент времени — набор целевых миров может изменяться в ходе исполнения программы. Если вместо целевого параметра в конструкторе резидента задан конструктор мира, целевым параметром считается экземпляр класса, построенный в результате доказательства этого конструктора мира. Если целевой параметр или заменяющий его конструктор мира в конструкторе резидента не заданы, целевым параметром считается предопределённый атрибут self.

Резидент решает следующие задачи и осуществляет следующие действия:

1. Построение множества списков, соответствующих различным целевым мирам резидента. Каждый такой список должен содержать все значения, возвращаемые в результате исполнения вызова функции (заданного в составе конструктора резидента) в рассматриваемом целевом мире.

2. Передача построенного множества списков владельцу резидента в составе (переключающего) потокового сообщения через защищающий порт резидента. В качестве потокового сообщения передаётся текущее значение целевого параметра в конструкторе резидента, в котором все целевые экземпляры классов заменены соответствующими им списками значений функции.

   В случае если некоторые списки ещё не построены, или если некоторые целевые процессы находятся в состоянии «неиспользуемый», «неудачный», «объявленный» или «сформированный», а также если в составе значения целевого параметра вместо некоторых миров присутствуют несвязанные переменные, вместо соответствующих списков значений функции в составе потокового сообщения передаются спейсеры #.

3. Постоянное слежение за состояниями целевых процессов. Повторное построение и передача списков значений функции после каждой фазы исполнения целевого процесса.

4. Приём новых значений через простые порты при изменении соответствующих общих переменных. Повторное построение и передача списков значений функции при получении новых значений через простые порты.

Вычисление значений функции в целевом мире допускается лишь в том случае, если целевой процесс уже обработал все полученные им потоковые сообщения, и, следовательно, не имеет несогласованных портов.

Перед началом вычисления значений функции осуществляется активизация некоторых портов целевого процесса в соответствии с правилами обработки прямых сообщений (см. раздел 7.4.2).

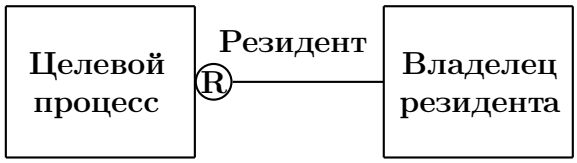
Для вычисления значений функции в целевом мире (временно) создаётся и доказывается с откатом актор Q. В случае если доказательство актора Q завершается исключительной ситуацией, (недостроенный) список значений функции теряется, и резидент осуществляет повторную попытку построить список значений функции.

Перед отправлением построенного списка значений функции осуществляется его упорядочение и сокращение с помощью удаления повторных элементов. В конкретной реализации языка должны быть заданы однозначные правила упорядочения термов (значений функций).

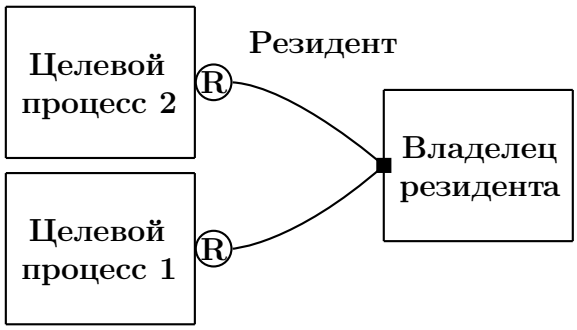
Примечание. Повторное построение списков значений функции резидента разрешается не производить в тех случаях, когда повторное исполнение функции не приведёт к получению новых значений.

Примечание. Рекомендуемые графические обозначения резидентов:

а) Резидент, которому соответствует один целевой мир.



б) Резидент, которому соответствуют несколько целевых миров.



Ссылки: активизация порта 5.2.2, актор 7.1, атом 6.1, атрибут 4.1.1, вызов функции 6.2.1, доказательство актора 6.3.1, защищающий порт 5.2.2, значение терма 3, инициализатор 4.1.2, исключительная ситуация 7.5, исполнение предиката 6.3.1, исполнение программы 4, исполнение процесса 5.2, конструктор мира 4.1.3, конструктор резидента 4.1.3, мир 5.1, неиспользуемый процесс 5.2.1, несвязанная переменная 3.1, несогласованный порт 5.2.2, неудачный процесс 5.2.1, обработка потокового сообщения 7.4.3, обработка прямого сообщения 7.4.2, общие переменные 7.2, объявленный процесс 5.2.1, откат 6.3.4, передача потокового сообщения 7.4.3, переключающие сообщения 7.4.1, переменная 2.1.1, порт 5.2.2, потоковые сообщения 7.4.3, простой порт 5.2.2, процесс 5.2, слот 5.1, состояние процесса 5.2.1, спейсер 3.1, список 3.2.2, сформированный процесс 5.2.1, терм 3, фаза 5.2, функция 6.1.3, целевой параметр 4.1.3, self 4.1.1.

5.4 Построение пространства поиска

Построение пространства поиска осуществляется в результате доказательства конструкторов экземпляров классов и реализуется посредством построения новых экземпляров классов.

Последовательность действий, создающих пространство поиска и слоты отдельного экземпляра класса, называется «формированием» экземпляра класса.

Построение экземпляра класса (создание нового экземпляра класса) включает следующие этапы:

1. Формирование экземпляра класса.

2. Доказательство предикатов goal() во всех сформированных на первом этапе мирах.

Этапы формирования экземпляра класса и доказательства предикатов goal не зависят друг от друга и относятся к разным этапам построения процесса. Выполнение этих операций осуществляется в соответствии со следующими правилами:

1. При построении экземпляров классов используются копии определений классов, отличающиеся от соответствующих определений классов тем, что все переменные в составе определений атрибутов этих классов заменяются новыми уникальными именами.

2. Каждая автоматически исполняемая подцель goal объявляется актором.

3. Исполнение предикатов goal осуществляется в произвольном порядке. При этом, однако, при равных прочих условиях, предикат goal в мире B всегда доказывается раньше, чем в мире A, если мир B является вложенным по отношению к миру A.

Ссылки: актор 7.1, атрибут 4.1.1, вложенность миров 5, исполнение предиката 6.3.1, класс 4.1, конструктор мира 4.1.3, мир 5.1, переменная 2.1.1, подцель доказательства 6.3.1, построение миров 5.4.1, построение процесса 5.4.1, слот 5.1, формирование миров 5.4.1, goal 5.4.1.

5.4.1 Исполнение конструкторов

Доказательство (исполнение) простого конструктора приводит к формированию нового экземпляра заданного класса C и включает следующие действия:

1. Построение пространства поиска экземпляра класса C.
   В состав пространства поиска включаются предложения самого класса C, затем (в соответствии с иерархией наследования) предложения его непосредственного предка D, предложения непосредственного предка класса D и так далее, пока не будет достигнут класс, не имеющий предка в иерархии наследования.

2. Построение слотов экземпляра класса.

Доказательство (исполнение) конструктора процесса приводит к построению нового процесса. В результате доказательства конструктора процесса, новый процесс устанавливается в состояние «объявленный». Исполнение конструктора процесса не приводит к построению пространства поиска созданного процесса. Построение соответствующего пространства поиска и слотов процесса осуществляется позже, в ходе «формирования» процесса.

Формирование процесса вызывается процессом, который является его создателем (см. раздел 7.4.3). Формирование процесса включает следующие действия:

1. Доказывается простой конструктор, заданный в составе конструктора процесса.

2. Определяются порты процесса, в соответствии с описателями портов, заданными в аргументах конструктора процесса и определениях соответствующих классов.

3. Проверяются текущие значения портов процесса и, в соответствии с правилами переключения состояний процесса «используемый» и «неиспользуемый» (см. раздел 5.2.1), процесс переводится в состояние «используемый сформированный» или в состояние «неиспользуемый».

4. Процессу автоматически посылается инициализирующее потоковое сообщение.

В случае если формирование процесса заканчивается аварийной ситуацией, вызывается встроенный обработчик ошибок.

Доказательство (исполнение) конструктора резидента приводит к построению нового резидента. Доказательство конструктора резидента включает следующие действия:

1. Переменная, созданная в качестве начального значения слота процесса-владельца, инициализатором которого является конструктор резидента (см. раздел 5.4.2), объявляется защищающим портом резидента.

2. Все остальные общие переменные, заданные в составе конструктора резидента, объявляются простыми портами резидента.

3. Новый резидент начинает функционировать.

Ссылки: владелец резидента 5.3, защищающий порт 5.2.2, значение порта 5.2.2, иерархия наследования 4.1, инициализатор 4.1.2, инициализирующее сообщение 7.4.3, используемый процесс 5.2.1, класс 4.1, конструктор 4.1.3, конструктор процесса 4.1.3, конструктор резидента 4.1.3, мир 5.1, начальное значение слота 5.4.2, неиспользуемый процесс 5.2.1, общие переменные 7.2, объявленный процесс 5.2.1, описатель порта 4.1.1, переменная 2.1.1, порт 5.2.2, построение слотов 5.4.2, потоковые сообщения 7.4.3, предложение 6, простой конструктор 4.1.3, простой порт 5.2.2, процесс 5.2, резидент 5.3, слот 5.1, создатель процесса 5.2, состояние процесса 5.2.1, сформированный процесс 5.2.1.

5.4.2 Построение слотов

Одновременно с созданием каждого слота, если для него задан соответствующий инициализатор, создаётся его «начальное» значение:

- Значение терма, если инициализатором слота является терм или конструктор резидента.

- Некоторый мир, если инициализатором является конструктор мира.

- Значение другого слота, если инициализатором является атрибут.

В качестве инициализатора каждого создаваемого слота используется значение соответствующего аргумента доказываемого конструктора или, если аргумент не задан, инициализатор в определении класса. При этом инициализаторы, заданные в определении любого класса C, отменяют («перекрывают») все инициализаторы соответствующих атрибутов в определениях предков класса C.

Построение значения слота, инициализатором которого является конструктор, вызывает доказательство этого конструктора.

Если инициализатором слота является конструктор резидента, создаётся специальная переменная, общая для резидента и его владельца. Эта переменная становится начальным значением слота процесса-владельца, инициализатором которого является названный конструктор резидента.

Если инициализатором слота является конструктор резидента, в составе которого задан некоторый конструктор мира, указанный конструктор мира также доказывается.

Слоты, не имеющие инициализаторов, а также слоты, (взаимно-) рекурсивно заданные в качестве своих собственных инициализаторов, получают в качестве начальных значений уникальные общие переменные.

Все переменные, создаваемые в составе значений слотов, также являются общими.

Если в составе аргумента доказываемого конструктора или в определении соответствующего класса некоторому атрибуту приписан описатель порта, действие описателя распространяется на все общие переменные в составе инициализатора соответствующего слота, исключая конструкторы и вызовы функций, входящие в этот инициализатор.

Описатели портов, заданные в составе конструктора, а также в определении соответствующих классов, перекрывают друг друга аналогично тому, как осуществляется перекрытие инициализаторов слотов. Перекрытие описателей портов происходит независимо от перекрытия инициализаторов слотов.

Пример. Перекрытие инициализаторов слотов и описателей портов.

Рассмотрим определение некоторых классов 'C', 'D' и 'E':

```
class 'E':
suspending: a = 21;
protecting: b = 25;
[
goal.
]
class 'D' specializing 'E':
protecting: a = 7;
[]
class 'C' specializing 'D':
b = 8;
[]
```

В результате доказательства простого конструктора ('C', a=X) будет построен некоторый мир, значения слотов которого a=X, b=8. Переменная X будет объявлена защищающим портом.

Ссылки: атрибут 4.1.1, владелец резидента 5.3, вызов функции 6.2.1, защищающий порт 5.2.2, значение слота 5.1, значение терма 3, инициализатор 4.1.2, класс 4.1, конструктор 4.1.3, конструктор мира 4.1.3, конструктор резидента 4.1.3, мир 5.1, общие переменные 7.2, описатель порта 4.1.1, переменная 2.1.1, порт 5.2.2, построение миров 5.4.1, простой конструктор 4.1.3, резидент 5.3, слот 5.1, терм 3, **class** 2.1.2, goal 5.4.1, **protecting** 2.1.2, **specializing** 2.1.2, **suspending** 2.1.2.

Глава 6

Предложения классов

Логические правила («предложения») состоят из заголовка и последовательности (возможно, пустой) подцелей. Предложения, в составе которых нет подцелей, называются «фактами».

предложение = атом [":–" конъюнкция] "."
конъюнкция = [конъюнкция ","] подцель

В предложении могут использоваться переменные и атрибуты. Область действия переменных ограничена пределами предложения. В составе предложения не допускается однократное использование переменных, отличных от «_».

Предложения, в которых используются метапеременные, называются метапредложениями.

Предложения каждого класса группируются в соответствии с их заголовками.

предложения = { предложение }

Предложения, не являющиеся метапредложениями, должны принадлежать одной группе («процедуре»), если совпадают имена и арность предикатных символов заголовков этих предложений. Процедуры, в свою очередь, также должны быть сгруппированы в соответствии с именами предикатных символов заголовков входящих в них предложений.

Считается, что метапредложения не входят в состав каких-либо процедур, однако такие метапредложения, в заголовке которых присутствует предикатный символ, и в качестве этого предикатного символа задан символ, должны быть сгруппированы с предложениями с таким же именем предикатного символа заголовков.

Примечание. Предложения, в заголовке которых задано объявление функции, группируются по общим правилам, вместе с другими предложениями класса. При этом арность предикатных символов заголовков этих предложений определяется без учёта терма или выражения, возвращаемого функцией.

Ссылки: атом 6.1, атрибут 4.1.1, класс 4.1, метапеременная 3, объявление функции 6.1.3, переменная 2.1.1, подцель 6.2, подцель предложения 6.2, символ 2.1.2, терм или выражение 3.2.1, функция 6.1.3.

6.1 Атомарные формулы

Атомарными формулами (атомами) в языке являются следующие обозначения:

атом =
    простой_атом |
    бинарное_отношение |
    объявление_функции

Ссылки: бинарное отношение 6.1.2, объявление функции 6.1.3, простой атом 6.1.1.

6.1.1 Простые атомы

Простой атом — это функтор с соответствующим количеством аргументов, недоопределённое множество или переменная:

простой_атом =
    функтор ["(" [термы_и_выражения ["*"]] ")"] |
    недоопределённое_множество |
    метапеременная

Последний аргумент атомарной формулы может быть помечен «*» только тогда, когда он является переменной. В этом случае атомарная формула обозначает предикат с переменным числом аргументов («предикат переменной арности»), а помеченная переменная — список аргументов, не определённых явно в составе атомарной формулы. Во время трансляции арность такого предиката неопределена, однако в ходе исполнения программы эта атомарная формула может быть

унифицирована, в общем случае, с атомом любой арности большей или равной $R - 1$, где R — количество аргументов, заданных в составе рассматриваемого предиката (включая помеченную переменную).

Переменные, помеченные «∗», а также переменные, используемые в качестве атомов и функторов, называются «метапеременными». Предикаты переменной арности, метапеременные, используемые в качестве атомов, а также атомы, в качестве функторов которых используются метапеременные, называются «метапредикатами» («метаатомами»).

Переменная в атомарной формуле подцели предложения может быть помечена «∗» лишь в том случае, если она таким же образом помечена в заголовке предложения и не является анонимной переменной «_». Для обозначения списка аргументов предиката переменной арности не разрешается использовать метафункторы.

Атомарная формула вида

$$A_0\{x_1{:}A_1,x_2{:}A_2,\ldots,x_n{:}A_n|Rest\}$$

эквивалентна

$$''(\{0{:}A_0,x_1{:}A_1,x_2{:}A_2,\ldots,x_n{:}A_n|Rest\}),$$

где '' — символ, состоящий из пустой цепочки графем.

Пример. Предложение, имитирующее правило 2-го порядка.

Для обозначения данных в примере используются недоопределённые множества, в состав которых входит признак чётности «is_even».

```
P{is_even:'any'|Rest}:-
    P{is_even:'yes'|Rest},
    P{is_even:'no'|Rest}.
```

Приведённое утверждение означает, что любой предикат P является истинным при чётных и нечётных значениях аргумента, если его истинность удаётся доказать отдельно для чётных и нечётных значений этого аргумента.

Ссылки: анонимная переменная 2.1.1, атом 6.1, графема 1, данные 3, заголовок предложения 6, исполнение программы 4, метапеременная 3, метафунктор 3, недоопределённое множество 3.2.3, переменная 2.1.1, подцель предложения 6.2, предложение 6, простой атом 6.1.1, символ 2.1.2, список 3.2.2, термы и выражения 3.2.1, унификация 3.3, функтор 3, '' 2.1.2.

6.1.2 Бинарные отношения

«Бинарным отношением» называется атомарная формула, состоящая из двух аргументов, соединённых оператором отношения:

бинарное_отношение =
терм_или_выражение оператор_отношения терм_или_выражение

В качестве знаков операций в бинарных отношениях используются имена встроенных предикатов '==' и ':=', а также некоторые знаки операций сравнения:

оператор_отношения =
"==" | ":=" | "<" | ">" | "<>" | "<=" | ">="

Бинарное отношение, в состав которого входит такой знак операции, эквивалентно обозначению вида

функтор(аргумент$_1$,аргумент$_2$),

где функтор — знак операции, заключённый в апострофы, аргумент$_1$ и аргумент$_2$ — операнды, стоящие соответственно слева и справа от знака операции.

Ссылки: атом 6.1, бинарное отношение 6.1.2, встроенный предикат 8, оператор отношения 6.1.2, терм или выражение 3.2.1, функтор 3, ':=' 8.1, '==' 3.3.

6.1.3 Объявления функций

Функциями называется разновидность предикатов, предназначенная для имитации подпрограмм-функций, возвращающих выходное значение. Определение функций осуществляется с помощью специальных синтаксических конструкций, называемых «объявлениями функций».

объявление_функции = простой_атом "=" терм_или_выражение

В качестве простых атомов в составе объявлений функций не разрешается использовать метапеременные.

В результате трансляции объявления функций преобразуются в предикаты. В ходе трансляции предложения, имитирующего объявление функции,

p(A$_1$,A$_2$,...,A$_n$) = E :– Конъюнкция. ,

оно преобразуется к виду

p(E,A_1,A_2,...,A_n):– Конъюнкция, S.

При этом все вызовы функций S, входящие в состав терма E, выносятся в конец предложения, после подцелей «Конъюнкция». Вызовы функций S всегда помещаются после любых других вызовов функций, вынесенных в конец предложения из его заголовка.

Ссылки: вызов функции 6.2.1, заголовок предложения 6, значение терма 3, метапеременная 3, объявление функции 6.1.3, подцель предложения 6.2, предложение 6, простой атом 6.1.1, терм 3, терм или выражение 3.2.1.

6.2 Подцели предложений

Подцелями предложения служат вызовы предикатов.

«Вызовом предиката» называется синтаксическая конструкция, определяющая экземпляр класса, в котором этот вызов должен быть исполнен, тип вызова (ближний или дальний), а также атомарную формулу вызова. Различаются ближние и дальние, а также простые и акторные вызовы предикатов.

Вызов предиката называется «дальним», если в подцели явным образом (с помощью переменной или атрибута) указан мир, в котором он должен быть исполнен. Если соответствующий мир не указан, вызов предиката называется «ближним». Исполнение ближнего вызова осуществляется в том же самом мире, в котором исполняется рассматриваемое предложение.

Акторными вызовами предикатов называются подцели предложений, определяющие акторы. Если про вызов предиката не сказано, что он является акторным, такой вызов называется (является) простым.

подцель =
    простая_подцель |
    бинарное_отношение |
    "[" [термы_и_выражения] "]" |
    "!"
простая_подцель =
    [[целевой_параметр] инфикс_подцели] простой_атом
инфикс_подцели = "?" | "<<" | "<–"

Если инфикс подцели равен «<<» или «<−», в качестве простого атома этой подцели не разрешается использовать метапеременные.

Подцель $[V_1,\ldots,V_k]$ обозначает встроенный управляющий оператор copy$(V_1,\ldots,V_k)$. Подцель «!» обозначает встроенный управляющий оператор отсечения '!'.

Пример. Правильно построенные предложения:

```
clause_1(M,N,J):–
    M * 2 + N < 7 – ? f(J),          –– простой ближний вызов
    console ? write("N=",N).         –– простой дальний вызов
clause_2(K,1,N,L*):–
    check(K, slot ? p(N) ),          –– простой ближний вызов
    ? p(N,7,L).                      –– простой ближний вызов
```

Ссылки: актор 7.1, атом 6.1, атрибут 4.1.1, бинарное отношение 6.1.2, встроенный оператор 8, инфикс подцели 6.2, исполнение предиката 6.3.1, исполнение предложения 6.3.2, метапеременная 3, мир 5.1, отсечение 8, переменная 2.1.1, предложение 6, простой атом 6.1.1, термы и выражения 3.2.1, целевой параметр 4.1.3, copy 8.2.

6.2.1 Вызовы функций

«Вызовом функции» называется синтаксическая конструкция, имитирующая вызов подпрограммы-функции.

В составе предложений разрешается использовать следующие вызовы функций:

```
вызов_функции_в_предложении =
    [ целевой_параметр ] "?" простой_атом |
    целевой_параметр "[" термы_и_выражения "]"
```

В качестве простых атомов в составе вызовов функций не разрешается использовать метапеременные.

Если в составе вызова функции целевой параметр не задан явно, целевым параметром считается предопределённый атрибут self.

В результате трансляции вызовы функций вида

$$W\ ?\ p(A_1,A_2,\ldots,A_n)$$

преобразуются в вызовы предикатов вида

$$W\ ?\ p(R,A_1,A_2,\ldots,A_n)\ ,$$

а вызовы функций вида

$$X\ [A_1,A_2,\ldots,A_n]$$

— в вызовы предикатов вида

$$X\ ?\ element(R,A_1,A_2,\ldots,A_n)\ ,$$

где R — некоторая уникальная переменная, обозначающая результат функции и помещаемая на место транслируемого вызова функции.

В случае если вызов функции используется в составе некоторой подцели SA, вызов предиката SB, соответствующий этому вызову функции, добавляется в состав предложения перед подцелью SA. При этом гарантируется, что подцель SB будет помещена перед любой другой подцелью, в состав которой войдёт переменная R, обозначающая результат рассматриваемого вызова функции, но после всех подцелей, находившихся перед подцелью SA в исходном предложении.

Если вызов функции используется в заголовке предложения вида

Заголовок:– Конъюнкция.

и не является составной частью другого вызова функции, вызов предиката S, соответствующий этому вызову функции, добавляется в состав предложения после конъюнкции подцелей «Конъюнкция»:

Заголовок:– Конъюнкция, S.

Если вызов функции FC1 в заголовке предложения входит в состав другого вызова функции FC2, в этом случае FC1 рассматривается как вызов функции, входящий в состав подцелей предложения, поставленных в соответствие вызову функции FC2.

Примечание. В соответствии с правилами исполнения вызова предиката, определёнными в разделе 6.3.1, вызовами функций считаются также такие подцели метапредложений, атомарная формула которых является метапеременной, при условии что рассматриваемое метапредложение поставлено в соответствие вызову функции.

Пример. Определение функции append.

Определение функции append, добавляющей элементы в конец списка,

```
append([ ],L) = L.
append([H|L1],L2) = [H | ?append(L1,L2)].
```

соответствует процедуре вида

```
append(L,[ ],L).
append(R0,[H|L1],L2):–
    append(R1,L1,L2),
    R0 == [H | R1].
```

Ссылки: атом 6.1, атрибут 4.1.1, вызов предиката 6.2, заголовок предложения 6, исполнение предиката 6.3.1, конъюнкция 6, метапеременная 3, метапредложение 6, объявление функции 6.1.3, переменная 2.1.1, подцель доказательства 6.3.1, подцель предложения 6.2, предложение 6, простой атом 6.1.1, процедура 6, список 3.2.2, термы и выражения 3.2.1, функция 6.1.3, целевой параметр 4.1.3, self 4.1.1, '==' 3.3.

6.2.2 Выражения

Выражение — это видоизменённый вызов функции:

```
выражение =
    [ выражение аддитивный_оператор ] слагаемое |
    выражение аддитивный_оператор терм
слагаемое =
    [ слагаемое мультипликативный_оператор ] множитель |
    слагаемое мультипликативный_оператор терм
множитель =   [ "–" ] "(" выражение ")"
```

Для построения выражений используется ограниченный набор знаков операций, в состав которого входят следующие математические символы:

```
аддитивный_оператор =   "+" | "–"
мультипликативный_оператор =   "*" | "/"
```

Выражение, построенное с помощью инфиксного знака операции, эквивалентно вызову функции вида

?функтор(аргумент$_1$,аргумент$_2$),

где функтор — знак операции, заключённый в апострофы, аргумент$_1$ и аргумент$_2$ — операнды, стоящие соответственно слева и справа от знака операции.

Выражение, построенное с помощью префиксного знака операции «−», эквивалентно вызову функции вида

?'−'(аргумент),

аргументом которой является операнд выражения.

Пример. Правильно построенные выражения:
1+H∗−(E)/4+(W+"A4"−319e0), 'f'∗X+(7−"t")−'r'

Ссылки: вызов функции 6.2.1, терм 3, функтор 3.

6.3 Стратегия управления

Стратегия управления Акторного Пролога («акторный механизм») является расширением стандартной стратегии управления («поиск слева направо в глубину с возвратом»), соответствующей текстуальному упорядочению процедур и вызовов предикатов. Отличиями акторного механизма от стандартной стратегии управления являются возможности повторного доказательства акторов, а также задержки исполнения подцелей.

Ссылки: актор 7.1, акторный механизм 7, вызов предиката 6.2, исполнение предиката 6.3.1, механизм задержки 6.3.3, повторные доказательства 7.1, процедура 6.

6.3.1 Исполнение вызова предиката

Общая схема исполнения вызова предиката (исполнения предиката) включает следующие действия:

1. Выбор предложения, заголовок которого:
   (a) Содержит предикатный символ, совпадающий с именем вызывающего предиката. В случае исполнения вызова функции, запрещается выбирать предложения, заголовки которых не являются объявлениями функций.

   (b) Является метапеременной.

   (c) Содержит метафунктор в качестве предикатного символа.

2. Если следом за выбранным предложением в рассматриваемом мире расположены ещё не исследованные предложения, в исполняемом процессе создаётся новая точка выбора, обозначающая поиск иных предложений, соответствующих условиям пункта 1.

3. Построение копии предложения, отличающейся от выбранного предложения тем, что все его переменные заменяются новыми уникальными именами, а все атрибуты заменяются значениями слотов мира, которому принадлежит предложение.

4. Исполнение построенного предложения.

При этом исполнение вызовов предикатов различных типов осуществляется в соответствии со следующими дополнительными правилами и исключениями:

1. Если значением целевого параметра в дальнем вызове предиката является спейсер #, исполнение подцели заканчивается успехом (без какого-либо связывания переменных).

2. Если значением целевого параметра в дальнем вызове предиката является элемент данных, исполнение подцели осуществляется непосредственно в том мире, в котором исполняется рассматриваемое предложение. При этом целевой параметр добавляется в исполняемый предикат в качестве дополнительного аргумента перед другими аргументам предиката, но после аргумента, обозначающего возвращаемое значение функции (в случае если исполняемая подцель является вызовом функции).

3. Исполнение дальних вызовов предикатов с инфиксами «<<» и «<−» всегда заканчивается успехом и заключается в подготовке соответствующих прямых сообщений, передача которых откладывается до (успешного) завершения рассматриваемой фазы исполнения процесса. Инфикс «<<» обозначает информационные прямые сообщения, а инфикс «<−» — переключающие прямые сообщения.

4. Исполнение дальних вызовов предикатов, не имеющих инфикса «<<» или «<−», в мирах, не принадлежащих процессу, в котором исполняется рассматриваемое предложение, невозможно и заканчивается неудачей.

5. Исполнение акторных вызовов предикатов осуществляется в соответствии с общей схемой, но при этом дополнительным необходимым условием успешного завершения доказательства любого актора P процесса G является существование производных значений общих переменных процесса G. Для того чтобы обеспечить существование требуемых производных значений, в момент (успешного) завершения доказательства предиката P осуществляется согласование акторов процесса G. Доказательство актора P считается успешным в том и только в том случае, если завершаются успехом исполнение соответствующего предиката, а также последующее согласование акторов, в противном случае происходит откат.

6. Если атомарная формула исполняемого вызова предиката в составе метапредложения является метапеременной, данная подцель считается вызовом функции, если (и только если) рассматриваемое метапредложение поставлено в соответствие вызову функции.

7. Действия, осуществляемые при исполнении предопределённых предикатов и встроенных управляющих операторов рассмотрены в главе 8.

Подцели копии предложения, построенной во время исполнения предиката, называются «подцелями доказательства».

Ссылки: актор 7.1, акторный вызов 6.2, атом 6.1, атрибут 4.1.1, встроенный оператор 8, вызов предиката 6.2, вызов функции 6.2.1, дальний вызов 6.2, данные 3, заголовок предложения 6, значение слота 5.1, значение терма 3, инфикс подцели 6.2, информационные сообщения 7.4.1, исполнение предложения 6.3.2, исполнение процесса 5.2, метапеременная 3, метапредложение 6, метафунктор 3, мир 5.1, общие переменные 7.2, объявление функции 6.1.3, откат 6.3.4, передача сообщения 7.4, переключающие сообщения 7.4.1, переменная 2.1.1, подцель предложения 6.2, предложение 6, предопределённый предикат 8, производные значения 7.2, процесс 5.2, прямые сообщения 7.4.2, связывание 3.3, согласование акторов 7.3, спейсер 3.1, фаза 5.2, функция 6.1.3, целевой параметр 4.1.3.

6.3.2 Исполнение предложения

Будем говорить, что переменная или слот имеют «задерживающее» («отключающее») значение, если они несвязаны или их значение равно спейсеру #.

Задержанными подцелями называются подцели доказательства, исполнение которых было отложено механизмом задержки исполнения подцелей. Списком задержанных подцелей называется вспомогательный список подцелей доказательства, используемый механизмом задержки исполнения подцелей.

Исполнение предложения включает:

1. Унификацию функтора и аргументов исполняемого вызова предиката с функтором и аргументами заголовка предложения или унификацию вызова предиката с метапеременной (если заголовком предложения является метапеременная). В случае если заголовок предложения является объявлением функции, а исполняемый вызов предиката не является вызовом функции, перед унификацией в начало списка аргументов исполняемого вызова добавляется фиктивный аргумент — анонимная переменная «_».

2. Пересмотр списка задержанных подцелей, осуществляемый механизмом задержки исполнения подцелей.

3. Исполнение соответствующих подцелей доказательства. При этом подцели доказательства, целевые параметры которых имеют задерживающие значения, пропускаются и добавляются в список задержанных подцелей.

Исполнение предложения заканчивается успехом тогда и только тогда, когда успехом заканчиваются все три названные операции.

В случае унификации функтора исполняемого вызова предиката с метафунктором, значением метафунктора становится соответствующий терм — символ. В случае унификации исполняемого вызова предиката с метапеременной, значением метапеременной становится соответствующий терм — символ или структура. Однако при использовании названных метафункторов и метаатомов в качестве предикатных символов и предикатов подцелей, они рассматриваются, соответственно, как правильно построенные имена предикатов и предикаты.

Вызов предопределённых предикатов и встроенных управляющих операторов с помощью метаатомов невозможен и всегда заканчивается неудачей.

Примечание. Метаатом в заголовке метапредложения может быть только символом или только структурой, потому что, в соответствии с семантикой языка, количество аргументов структуры не может быть меньше единицы.

Ссылки: анонимная переменная 2.1.1, встроенный оператор 8, вызов предиката 6.2, вызов функции 6.2.1, заголовок предложения 6, значение переменной 3.1, значение терма 3, исполнение предиката 6.3.1, метаатом 6.1.1, метапеременная 3, метапредложение 6, метафунктор 3, механизм задержки 6.3.3, несвязанная переменная 3.1, объявление функции 6.1.3, переменная 2.1.1, пересмотр списка задержанных 6.3.3, подцель доказательства 6.3.1, подцель предложения 6.2, предложение 6, предопределённый предикат 8, символ 2.1.2, слот 5.1, спейсер 3.1, список 3.2.2, структура 3.2.1, терм 3, унификация 3.3, функтор 3, целевой параметр 4.1.3.

6.3.3 Механизм задержки исполнения

Механизмом задержки исполнения подцелей называется вспомогательная стратегия управления, откладывающая исполнение выделенных подцелей до тех пор, пока не будет вычислена некоторая информация, необходимая для корректного исполнения этих подцелей.

Пересмотр списка задержанных подцелей осуществляется следующим образом:

1. Элементы списка просматриваются в том порядке, в котором они были в него добавлены.

2. При обнаружении каждого элемента списка, значение целевого параметра которого не является задерживающим, найденная подцель исключается из рассматриваемого списка и исполняется.

Считается, что на каждой фазе исполнения процесса используется новый список задержанных подцелей. В начале фазы список задержанных подцелей является пустым.

Ссылки: задерживающие значения 6.3.2, значение терма 3, исполнение предиката 6.3.1, подцель доказательства 6.3.1, процесс 5.2, список 3.2.2, список задержанных подцелей 6.3.2, фаза 5.2, целевой параметр 4.1.3.

6.3.4 Откат программы

Откатом называется возобновление исполнения процесса, начиная с последней (неустранённой оператором отсечения) точки выбора. Откат выполняется автоматически в случае неудачи какой-либо операции, осуществляемой в ходе исполнения предложения.

В результате отката программы осуществляется восстановление состояний акторов процесса, в котором произошёл откат, на момент прохождения последней (неустранённой оператором отсечения) точки выбора (в том числе отмена всех связываний и сцеплений переменных, произошедших в акторах с момента прохождения этой точки, отмена нейтрализации и повторных доказательств акторов, а также отмена всех изменений, внесённых в список задержанных подцелей).

В результате отката отменяются все прямые сообщения, подготовленные после прохождения последней неустранённой точки выбора для передачи из рассматриваемого процесса.

Ссылки: актор 7.1, исполнение предложения 6.3.2, исполнение процесса 5.2, нейтрализация актора 7.1, отсечение 8, передача прямого сообщения 7.4.2, переменная 2.1.1, повторные доказательства 7.1, предложение 6, программа 4, процесс 5.2, прямые сообщения 7.4.2, связывание 3.3, состояние актора 7.1, список задержанных подцелей 6.3.2, сцепление переменных 3.3.

Глава 7

Акторы и повторные доказательства

Повторное доказательство акторов в Акторном Прологе автоматически поддерживает корректность логического вывода при использовании разрушающего присваивания и параллельных процессов.

Ссылки: актор 7.1, повторные доказательства 7.1, процесс 5.2, разрушающее присваивание 8.1.

7.1 Акторы

Актором называется подцель доказательства, соответствующая акторному вызову предиката.

Актор Q называется «вложенным» по отношению к актору P, если эти акторы принадлежат одному процессу, и доказательство актора Q, результаты которого в данный момент не отменены, происходит (произошло) в ходе доказательства актора P.

Нейтрализацией актора называется отмена всех результатов его доказательства, за исключением результатов доказательства вложенных по отношению к нему акторов.

Повторным доказательством актора называется повторение доказательства актора с самого начала.

Актор может находиться в одном из трёх состояний:

1. «активный» актор;

2. «доказанный» актор;

3. «нейтральный» актор.

Актор P, доказательство которого происходит (произошло) в ходе некоторой фазы F исполнения процесса G, считается (называется) «активным» с момента начала его доказательства до завершения рассматриваемой фазы F. В случае успешного завершения доказательства актора P, а также успешного завершения фазы F, актор P считается «доказанным» с момента завершения фазы F до его (возможной) нейтрализации на одной из последующих фаз исполнения процесса G.

Актор, предыдущее доказательство которого отменено, а повторное доказательство ещё не началось, называется «нейтральным». Перевод актора в активное состояние называется «активизацией» актора.

Ссылки: акторный вызов 6.2, вызов предиката 6.2, доказательство актора 6.3.1, исполнение процесса 5.2, подцель доказательства 6.3.1, принадлежать процессу 5.2, процесс 5.2, фаза 5.2.

7.2 Общие переменные

Будем говорить, что актор P «использует» переменную V (или что переменная V «соответствует», «принадлежит» актору P), если переменная V входит в состав подцели доказательства P или какой-либо другой подцели доказательства Q, построенной в ходе:

- текущего доказательства подцели P, если исполнение актора ещё не закончено,
- последнего завершившегося успехом доказательства актора P, если исполнение актора закончено, и он не является нейтральным,

не считая тех подцелей доказательства Q, которые были построены в ходе доказательства акторов, вложенных по отношению к P.

«Общей» называется переменная, которая используется (или может быть использована) несколькими акторами. Каждый актор хранит свои собственные («локальные») значения общих переменных.

Актуальными значениями общих переменных некоторого процесса называются значения, которые можно получить, унифицировав локальные значения всех общих переменных, соответствующих активным акторам этого процесса.

Во время унификации термов (в ходе исполнения процесса G) происходит замена тех (и только тех) вхождений переменных, которые соответствуют активным акторам (процесса G). Таким образом, локальные значения общих переменных любого активного актора процесса всегда равны актуальным значениям общих переменных этого процесса.

Производными значениями общих переменных некоторого процесса называются значения, которые можно получить (если они существуют), унифицировав локальные значения общих переменных всех активных и доказанных акторов этого процесса. В случае если общая переменная процесса (например, некоторый порт процесса) не соответствует ни одному из активных или доказанных акторов, её производным значением считается анонимная переменная «_».

Акторы процесса считаются согласованными между собой, если:

1. Все акторы, принадлежащие процессу, хотя бы один раз были доказаны.

2. Существуют производные значения общих переменных этого процесса.

«Фиксированными» значениями общих переменных называются значения этих переменных, в составе которых все несвязанные переменные заменены спейсером #. «Фиксированием» терма называется заменена спейсером всех несвязанных переменных в составе терма. В соответствии с семантикой Акторного Пролога, фиксирование не распространяется на переменные в составе миров, являющихся компонентами фиксируемого терма.

Глобальными операциями (с общими переменными) называются операции, в которых используются локальные значения, принадлежащие активным и доказанным акторам некоторого процесса — сопоставление локальных значений общих переменных, актуализация производных значений общих переменных, передача потоковых сообщений из процесса.

В случае если некоторая общая переменная используется для передачи потоковых сообщений между процессами, в соответствие этой переменной ставятся:

1. «Глобальное» значение — некоторый терм.

2. Сорт текущего глобального значения — вспомогательное логическое значение: «защищённое» или «незащищённое».

3. Производитель текущего глобального значения — процесс, построивший текущее глобальное значение.

Эти атрибуты являются едиными для всех процессов, передающих и принимающих потоковые сообщения через рассматриваемую переменную.

Считается, что изначально глобальное значение общей переменной равно пустому значению сорта «незащищённое», производителем которого является некоторый (уникальный) процесс, не совпадающий ни с одним из процессов программы.

Ссылки: активные акторы 7.1, актор 7.1, актуализация 8.2, анонимная переменная 2.1.1, вложенные акторы 7.1, доказанный актор 7.1, доказательство актора 6.3.1, значение переменной 3.1, исполнение предиката 6.3.1, исполнение процесса 5.2, мир 5.1, нейтральные акторы 7.1, несвязанная переменная 3.1, обработка потокового сообщения 7.4.3, передача потокового сообщения 7.4.3, переменная 2.1.1, подцель доказательства 6.3.1, порт 5.2.2, потоковые сообщения 7.4.3, принадлежать процессу 5.2, программа 4, процесс 5.2, пустое значение 7.4.3, сопоставление локальных значений 7.3.1, спейсер 3.1, терм 3, унификация 3.3.

7.2.1 Построение общих переменных

Общие переменные создаются автоматически в составе значений слотов во время формирования экземпляров классов.

При построении общих переменных выполняются следующие правила:

1. Если в составе инициализатора аргумента конструктора (явным образом) задан атрибут, то в качестве соответствующего ему значения слота берётся начальное значение этого слота (созданное во время построения слота), вместе с соответствующими ему общими переменными.

2. Если в определении атрибутов класса в составе некоторого инициализатора аргумента некоторого конструктора явным образом задана некоторая переменная X, то в качестве значения X берётся соответствующая ей общая переменная, созданная в ходе построения слотов экземпляра класса.

Примечание. Указанные правила построения общих переменных выполняются даже в том случае, если значение рассматриваемого слота или переменной ранее уже было конкретизировано в каких-либо акторах.

Ссылки: актор 7.1, атрибут 4.1.1, значение переменной 3.1, значение слота 5.1, инициализатор 4.1.2, конструктор 4.1.3, мир 5.1, начальное значение слота 5.4.2, общие переменные 7.2, переменная 2.1.1, построение слотов 5.4.2, слот 5.1, формирование миров 5.4.1.

7.3 Согласование акторов процесса

Согласованием акторов процесса G называются действия, осуществляемые для обеспечения согласованности акторов процесса. Согласование акторов процесса необходимо для того, чтобы обеспечить существование производных значений общих переменных этого процесса.

Согласование акторов включает:

1. сопоставление локальных значений общих переменных акторов процесса;

2. повторное доказательство акторов, нейтрализованных в ходе проведённого сопоставления локальных значений.

Ссылки: актор 7.1, нейтрализация актора 7.1, общие переменные 7.2, повторные доказательства 7.1, производные значения 7.2, процесс 5.2, согласованность акторов 7.2, сопоставление локальных значений 7.3.1.

7.3.1 Сопоставление локальных значений

На первом этапе согласования акторов сопоставляются локальные значения общих переменных, соответствующие различным акторам процесса G.

Сопоставление локальных значений общих переменных осуществляется следующим образом:

1. Вычисляются фиксированные актуальные значения общих переменных процесса G.

2. Нейтрализуются все акторы процесса G, находящиеся в состоянии «доказанный», локальные значения которых не могут быть унифицированы с фиксированными актуальными значениями процесса.

Порядок нейтрализации акторов в языке не определён.

Ссылки: актор 7.1, актуальные значения 7.2, доказанный актор 7.1, локальные значения 7.2, нейтрализация актора 7.1, общие переменные 7.2, процесс 5.2, согласование акторов 7.3, унификация 3.3, фиксированное значение 7.2.

7.3.2 Исполнение повторных доказательств

После сопоставления локальных значений общих переменных автоматически вызывается повторное доказательство всех нейтральных акторов процесса G, за исключением акторов-представителей портов процесса (см. свойства акторов-представителей в разделе 5.2.2).

Порядок исполнения повторных доказательств акторов в языке не определён.

Согласование акторов считается успешным в том и только в том случае, если завершаются успехом все повторные доказательства.

Примечание. В результате повторного доказательства недетерминированного актора, могут возникать новые точки выбора.

Ссылки: актор 7.1, актор-представитель 5.2.2, исполнение предиката 6.3.1, локальные значения 7.2, нейтральные акторы 7.1, общие переменные 7.2, повторные доказательства 7.1, порт 5.2.2, процесс 5.2, согласование акторов 7.3, сопоставление локальных значений 7.3.1.

7.4 Согласование процессов

Согласованием процессов называются действия, осуществляемые для обеспечения согласованности процессов. Согласование процессов происходит с помощью обмена асинхронными сообщениями.

Сообщением называется некоторое количество информации, передаваемое между процессами, представляющее для них единое целое.

Действия, реализующие распространение информации из одного процесса в другие, называются передачей сообщений. В общем случае, передача сообщений из некоторого процесса может осуществляться каждый раз после завершения очередной фазы его исполнения.

Действия, осуществляемые процессом в случае получения сообщения, называются обработкой сообщения. Обработка сообщения является отдель-

ной фазой исполнения процесса. Процесс не принимает и не посылает никакие сообщения до завершения очередной фазы своего исполнения.

Ссылки: исполнение процесса 5.2, процесс 5.2, согласованность процессов 5.2, фаза 5.2.

7.4.1 Классификация сообщений

В языке различаются прямые и потоковые, а также переключающие и информационные сообщения.

Прямые и потоковые сообщения имеют следующие принципиальные отличия:

1. Прямые сообщения передаются непосредственно от одного процесса к другому (в виде дальнего вызова предиката), а потоковые сообщения — от одного процесса ко многим (в виде значения общей переменной).

2. Прямые сообщения никогда не теряются при передаче, в то время как потоковые сообщения, которые процесс ещё не успел обработать, могут быть заменены более новой информацией.

Отличие переключающих и информационных сообщений состоит в том что:

1. В результате обработки переключающего сообщения процесс может перейти в состояние «доказанный», «неудачный» или (остаться в состоянии) «сформированный», в то время как после обработки информационного сообщения процесс всегда оказывается в состоянии «доказанный».

2. В отличие от переключающих сообщений, обработка информационных сообщений процессом откладывается до тех пор, пока он не окажется в состоянии «доказанный».

В случае если обработка переключающего сообщения завершилась неудачей или исключительной ситуацией, процесс переходит в состояние «неудачный» или — если перед началом обработки сообщения он находился в состоянии «сформированный» — в состояние «сформированный». В случае если неудачей или исключительной ситуацией завершилась обработка информационного сообщения, происходит «поглощение» сообщения:

состояние процесса восстанавливается на момент, предшествовавший обработке сообщения, а само сообщение отменяется.

В языке используются переключающие потоковые, а также прямые информационные и прямые переключающие сообщения.

Порядок обработки сообщений в языке не определён, однако в случае если процессу необходимо обработать несколько сообщений, прямые сообщения всегда обрабатываются после потоковых, а информационные, при прочих равных условиях, после переключающих. Если процесс, получивший сообщения является целевым процессом некоторого резидента, вычисление значений функции резидента, при прочих равных условиях, осуществляется после обработки переключающих сообщений, но до обработки информационных сообщений. Гарантируется, что при условии возможности обработки сообщения, оно обязательно будет обработано через некоторый конечный промежуток времени.

Ссылки: дальний вызов 6.2, доказанный процесс 5.2.1, значение переменной 3.1, значение терма 3, исключительная ситуация 7.5, неудачный процесс 5.2.1, обработка потокового сообщения 7.4.3, обработка сообщения 7.4, передача прямого сообщения 7.4.2, потоковые сообщения 7.4.3, процесс 5.2, прямые сообщения 7.4.2, резидент 5.3, сообщение 7.4, состояние процесса 5.2.1, сформированный процесс 5.2.1, функция 6.1.3, целевой процесс 5.3.

7.4.2 Прямые сообщения

Прямым сообщением называется сообщение, реализующее исполнение дальнего вызова предиката из одного процесса в другом.

Передача прямых сообщений, подготовленных в результате исполнения дальних вызовов предикатов в течение некоторой фазы F исполнения процесса G, происходит в случае успешного завершения рассматриваемой фазы F.

Перед передачей прямых сообщений все несвязанные переменные в их составе заменяются соответствующими фиксированными производными значениями процесса G.

Обработка прямого сообщения (принимающим) процессом H включает следующие действия:

1. Осуществляется активизация всех портов S, текущее значение D которых не является пустым, а производитель текущего значения не ра-

вен H, таких что:

(a) порт S относится к сорту «отключающий»;

(b) порт S относится к сорту «простой», а сорт текущего значения D равен «защищённое».

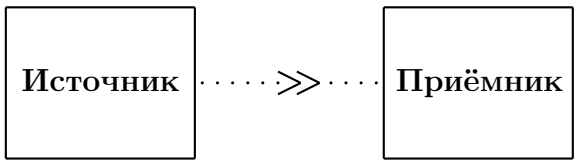
2. В соответствующем мире процесса H, указанном в дальнем вызове предиката P в составе обрабатываемого сообщения, (временно) создаётся и доказывается новый актор Q, соответствующий вызову предиката P. В частности, если процесс H находится в состоянии «сформированный», то в момент (успешного) завершения доказательства предиката P, во всех мирах этого процесса, сформированных в ходе исполнения конструктора процесса H, (для согласования акторов процесса H) создаются и доказываются новые акторы, представленные акторными вызовами предиката goal.

3. В случае успешного завершения доказательства актора Q, обработка сообщения считается успешно завершённой. В случае если доказательство актора Q завершилось неудачей или исключительной ситуацией, обработка сообщения прекращается.

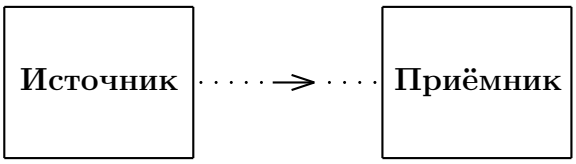
4. После завершения обработки сообщения, актор Q прекращает существование.

Обработка прямых сообщений процессом допускается лишь в том случае, если он уже обработал все полученные им потоковые сообщения, и, следовательно, не имеет несогласованных портов.

Примечание. Рекомендуемые графические обозначения прямых сообщений:
а) Информационное прямое сообщение.



б) Переключающее прямое сообщение.



Ссылки: активизация порта 5.2.2, актор 7.1, вызов предиката 6.2, дальний вызов 6.2, доказательство актора 6.3.1, значение порта 5.2.2, информационные сообщения 7.4.1, исключительная ситуация 7.5, исполнение конструктора 5.4.1, исполнение предиката 6.3.1, исполнение процесса 5.2, мир 5.1, несвязанная переменная 3.1, несогласованный порт 5.2.2, обработка потокового сообщения 7.4.3, отключающий порт 5.2.2, переключающие сообщения 7.4.1, переменная 2.1.1, порт 5.2.2, потоковые сообщения 7.4.3, производитель значения порта 5.2.2, производные значения 7.2, простой порт 5.2.2, процесс 5.2, пустое значение 7.4.3, согласование акторов 7.3, сообщение 7.4, сорт значения порта 5.2.2, сорт порта 5.2.2, состояние процесса 5.2.1, сформированный процесс 5.2.1, фаза 5.2, фиксированное значение 7.2, формирование миров 5.4.1, goal 5.4.1.

7.4.3 Потоковые сообщения

Потоковыми сообщениями называются сообщения, реализующие передачу производных значений общих переменных из одного процесса в другие. Потоковые сообщения передаются и принимаются через порты процессов.

Каждому потоковому сообщению ставятся в соответствие:

1. Значение потокового сообщения — некоторый терм.
2. Сорт потокового сообщения — вспомогательное логическое значение: «защищённое» или «незащищённое».
3. Производитель потокового сообщения — процесс, построивший значение потокового сообщения.

Существуют следующие вспомогательные разновидности потоковых сообщений:

1. «Пустые» сообщения — потоковые сообщения, значения которых равны анонимной переменной «_». Значения пустых сообщений называются «пустыми» значениями.

2. «Отключающие» сообщения — потоковые сообщение, значения которых являются задерживающими.

Потоковые сообщения, отличные от пустых, называются «непустыми». Значения непустых потоковых сообщений называются «непустыми» значениями.

Различаются два сорта потоковых сообщений — «защищённое» и «незащищённое». Использование двух разновидностей сообщений необходимо для управления передачей потоковых сообщений между процессами. В ходе передачи потоковых сообщений из некоторого процесса G незащищённое сообщение не может изменить глобальное значение, переданное в виде защищённого сообщения из какого-либо другого процесса (отличного от G).

Потоковое сообщение является защищённым (относится к сорту «защищённое»), если оно непустое и было отправлено (передано) через защищающий порт. В остальных случаях потоковое сообщение является незащищённым (относится к сорту «незащищённое»). В частности, сорт пустого сообщения всегда «незащищённое». В дальнейшем мы будем использовать специальное обозначение — функцию «сорт потокового сообщения», реализующую описанные выше правила и зависящую от значения A сообщения и сорта порта R: sort_of_flow_message(A,R).

Передача непустых переключающих потоковых сообщений из процесса осуществляется каждый раз при переходе этого процесса в состояние «доказанный», за исключением фаз процесса, закончившихся поглощением принятого процессом сообщения.

Передача пустых сообщений из процесса осуществляется каждый раз при переходе этого процесса из состояния «доказанный» в состояние «неудачный» или «неиспользуемый». Передача пустых сообщений из процесса называется «освобождением» общих переменных.

Будем придерживаться следующих обозначений: sort(X) — «сорт X», state(X) — «состояние X», creator(X) — «производитель, построивший X», X==Y — «возможна унификация термов X и Y», X<>Y — «X не равно Y», X:=Y — «X получает значение Y».

Передача потоковых сообщений осуществляется следующим образом:

1. После окончания очередной фазы исполнения некоторого процесса G строятся значения отправляемых потоковых сообщений:

   (a) фиксированные производные значения процесса, если отправляются непустые сообщения, или

   (b) пустые значения, если отправляются пустые сообщения.

2. Каждое построенное значение A сопоставляется с текущим значением B порта R процесса G, через который оно должно быть передано. В соответствии с результатами сопоставления осуществляются следующие действия:

   (a) Если A==B и sort_of_flow_message(A,R)=sort(B), устанавливается state(R):= «согласованный», передача рассматриваемого потокового сообщения отменяется.

   (b) Если sort_of_flow_message(A,R)=«незащищённое», sort(B)=«защищённое» и creator(B)<>G, передача рассматриваемого потокового сообщения отменяется.

   (c) В остальных случаях устанавливаются B:= A, sort(B):= sort_of_flow_message(A,R), creator(B):= G, state(R):= «согласованный».

3. Текущее значение B каждого порта R процесса G сопоставляется с текущим глобальным значением V этой переменной. В соответствии с результатами сопоставления осуществляются следующие действия:

   (a) Если B==V и sort(B)=sort(V), передача рассматриваемого потокового сообщения отменяется.

   (b) Если sort(B)=«незащищённое», sort(V)=«защищённое» и creator(V)<>G, передача рассматриваемого потокового сообщения отменяется.

   (c) В остальных случаях устанавливаются V:= B, sort(V):= sort(B), creator(V):= G.

4. Для каждого порта S каждого процесса H, таких что H<>G и V является переменной (портом S) процесса H, осуществляются следующие действия:

   (4i) Сопоставляются текущее глобальное значение V и текущее значение D порта S. В соответствии с результатами сопоставления осуществляются следующие действия:

(a) Если V==D, sort(V)=sort(D) и creator(V)=creator(D), никакие действия не осуществляются.

(b) Если V==D, sort(V)=sort(D), но creator(V)<>creator(D), устанавливается creator(D):= creator(V).

(c) Если sort(V)=«незащищённое», sort(D)=«защищённое» и creator(D)=H, никакие действия не осуществляются.

(d) В остальных случаях устанавливаются D:= V, sort(D):= sort(V), creator(D):= creator(V), state(S):= «несогласованный».

Последовательности действий, осуществляемых на этапах (2), (3), (4i) не могут быть прерваны никакими операциями, осуществляемыми другими процессами, использующими те же операнды.

Признаком получения некоторым процессом H потокового сообщения является наличие у процесса несогласованного порта. В случае получения процессом потокового сообщения, он осуществляет обработку этого сообщения.

Если процесс получил задерживающее значение через некоторый отключающий порт, обработка любых сообщений этим процессом откладывается, а процесс переводится в состояние «неиспользуемый». В дальнейшем, процесс будет автоматически возвращён в состояние «используемый», как только значения всех его отключающих портов перестанут быть задерживающими (см. свойства отключающих портов в разделе 5.2.2).

При переходе процесса H из состояния «неиспользуемый» в состояние «сформированный», автоматически вызывается формирование процессов, которые ещё не были сформированы, создателем которых является процесс H.

Если процессу необходимо обработать несколько переключающих потоковых сообщений, осуществляется их одновременная обработка («интерференция» потоковых сообщений).

Обработка потоковых сообщений некоторым процессом H включает следующие действия:

1. Осуществляется активизация всех портов S, текущее значение D которых не является пустым, а производитель текущего значения не равен H, таких что:

   (a) порт S является несогласованным, за исключением защищающих портов S, значения которых являются незащищёнными;

   (b) порт S относится к сорту «отключающий»;

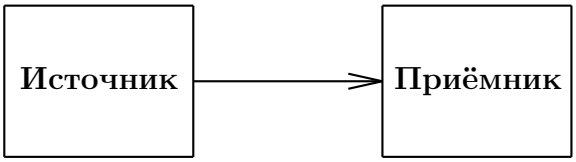
   (c) порт S относится к сорту «простой», а сорт текущего значения D равен «защищённое».

2. Все порты процесса H приводятся в согласованное состояние.

3. Осуществляется согласование акторов процесса. В частности, если процесс H находится в состоянии «сформированный», во всех мирах этого процесса, сформированных в ходе исполнения конструктора процесса H, создаются и доказываются новые акторы, представленные акторными вызовами предиката goal.

Специальной разновидностью переключающих потоковых сообщений является «инициализирующее» сообщение, автоматически получаемое каждым новым процессом. Инициализирующее сообщение считается полученным процессом независимо от того, имеет ли он какие-либо порты, а также от состояния этих портов. При этом, однако, осуществляется интерференция инициализирующего сообщения с любыми другими переключающими потоковыми сообщениями, а обработка инициализирующего сообщения происходит по общим правилам обработки потоковых сообщений.

Примечание. Рекомендуемое графическое обозначение для переключающих потоковых сообщений:



Ссылки: активизация порта 5.2.2, актор 7.1, анонимная переменная 2.1.1, глобальные значения 7.2, доказанный процесс 5.2.1, доказательство актора 6.3.1, задерживающие значения 6.3.2, защищающий порт 5.2.2, значение порта 5.2.2, значение терма 3, исполнение конструктора 5.4.1, исполнение процесса 5.2, используемый процесс 5.2.1, конструктор процесса 4.1.3, мир 5.1, неиспользуемый процесс 5.2.1, несогласованный порт 5.2.2, неудачный процесс 5.2.1, обработка сообщения 7.4, общие переменные 7.2, отключающий порт 5.2.2, переключающие сообщения 7.4.1, переменная 2.1.1, поглощение сообщений 7.4.1, порт 5.2.2, производитель значения порта 5.2.2, производные значения 7.2, простой порт 5.2.2, процесс 5.2, согласование акторов 7.3, согласованный порт 5.2.2, создатель

процесса 5.2, сообщение 7.4, сорт глобального значения 7.2, сорт значения порта 5.2.2, сорт порта 5.2.2, состояние порта 5.2.2, состояние процесса 5.2.1, сформированный процесс 5.2.1, терм 3, унификация 3.3, фаза 5.2, фиксированное значение 7.2, формирование миров 5.4.1, формирование процесса 5.4.1, goal 5.4.1.

7.5 Исключительные ситуации

Исключительной ситуацией называется аварийное состояние вычислительного процесса, зарегистрированное во время исполнения программы. Обработкой исключительной ситуации называются действия, осуществляемые программой в случае возникновения исключительной ситуации.

Любые исключительные ситуации, возникающие в ходе доказательства, считаются локальными по отношению к самому внутреннему из вложенных акторов, в котором эта ситуация возникла. Доказательство такого актора останавливается и объявляется неудачным, все его результаты отменяются таким образом, как это происходит при откате программы.

После прекращения доказательства актора P, в том же самом мире, где был исполнен соответствующий акторный вызов, автоматически вызывается предикат

alarm(Exception),

аргументом которого является «обозначение» обрабатываемой исключительной ситуации — неотрицательное целое число или символ. Если доказательство предиката alarm заканчивается успехом, обработка исключительной ситуации завершается, после чего происходит откат из точки вызова доказательства актора P. Если доказательство заканчивается неудачей, происходит вызов исключительной ситуации с тем же обозначением на следующем уровне вложенности акторов. Если в ходе доказательства предиката alarm возникает новая исключительная ситуация, её обработка также осуществляется на следующем уровне вложенности акторов.

Если в результате возникновения исключительной ситуации прекращается доказательство некоторого актора процесса G, не являющегося вложенным по отношению к какому-либо другому актору этого процесса, управление передаётся встроенному обработчику ошибок.

Действия встроенного обработчика ошибок должны быть определены в конкретной реализации языка. При этом, однако, встроенному обработчику ошибок не разрешается самостоятельно приостанавливать или прекращать исполнение каких-либо процессов программы.

Исключительная ситуация может быть вызвана из программы с помощью встроенного управляющего оператора

break(Exception),

где Exception — обозначение исключительной ситуации. Если значение аргумента оператора не является обозначением исключительной ситуации, или если в качестве аргумента использована несвязанная переменная, исполнение оператора завершается неудачей. Допускается использование оператора break без аргумента, в этом случае обозначением исключительной ситуации считается константа 0 (ноль).

Обозначения исключительных ситуаций, вызываемых на системном уровне (за пределами текста программы), должны быть определены в конкретной реализации языка.

Пример. Вызов исключительной ситуации.

```
class 'Example' specializing 'TextPage':
[
goal:–
    break('name_of_exception').
alarm(Code):–
    write("Exception: ",Code).
]
```

В результате доказательства предиката goal будет напечатано:

Exception: name_of_exception

Ссылки: актор 7.1, акторный вызов 6.2, вложенные акторы 7.1, встроенный оператор 8, встроенный предикат 8, доказательство актора 6.3.1, значение терма 3, исполнение предиката 6.3.1, исполнение программы 4, исполнение процесса 5.2, константа 3.1, мир 5.1, несвязанная переменная 3.1, откат 6.3.4, программа 4, процесс 5.2, символ 2.1.2, число 3.1, **class** 2.1.2, goal 5.4.1, **specializing** 2.1.2.

Глава 8

Встроенные предикаты и операторы

Встроенными предикатами языка являются goal(), alarm(E), ''(Set) и element(Value,I_1,...,I_k), определяемые в тексте программы, а также предопределённые предикаты:

'=='(V_1,...,V_k)	—	унифицировать термы;
':='(V_1,...,V_k)	—	разрушающее присваивание;
true[(...)]	—	истина;
fail	—	ложь (неудача).

Кроме того, в языке определены встроенные управляющие операторы, использование которых может нарушить полноту программы относительно её декларативной семантики:

copy(V_1,...,V_k)	—	актуализация производных значений;
'!'	—	отсечение;
break[(E)]	—	вызов исключительной ситуации;
spypoint(...)	—	обращение к отладчику.

Встроенный оператор отсечения устраняет все неисследованные пути (точки выбора), которые встретились с момента начала исполнения предиката, в соответствие которому было поставлено предложение, содержащее оператор.

Результаты исполнения оператора «обращение к отладчику» должны быть определены в конкретной реализации языка.

В программе не допускается определение предикатов, имена которых совпадают с именами предопределённых предикатов и встроенных управляющих операторов. Не разрешается использование таких имён в качестве предикатных символов в акторных и дальних вызовах.

Неверное число аргументов в предопределённых предикатах и встроенных управляющих операторах является синтаксической ошибкой.

Примечание. Обозначение оператора '!' с помощью ограничителя «!» рассмотрено в разделе 6.2.

Пример. Использование оператора отсечения.

Рассмотрим поведение фрагмента программы

```
goal:-
    write("<1>"),
    subgoal_a,
    write("<7>").
goal:-
    write("<8>").

subgoal_a:-
    write("<2>"),
    subgoal_b, !,                    -- отсечение
    write("<4>"),
    fail.
subgoal_a:-
    write("<6>").

subgoal_b:-
    write("<3>").
subgoal_b:-
    write("<5>").
```

Если убрать оператор отсечения, программа напечатает:

```
<1><2><3><4><5><4><6><7>
```

При наличии оператора отсечения будет напечатано:

<1><2><3><4><8>

Ссылки: акторный вызов 6.2, актуализация 8.2, вызов предиката 6.2, дальний вызов 6.2, исключительная ситуация 7.5, исполнение предиката 6.3.1, предложение 6, программа 4, разрушающее присваивание 8.1, унифицировать 3.3, alarm 7.5, break 7.5, copy 8.2, element 6.2.1, goal 5.4.1, ! 6.2, '' 2.1.2, ':=' 8.1, '==' 3.3.

8.1 Корректное разрушающее присваивание

Разрушающим присваиванием в Акторном Прологе называется изменение производных значений общих переменных процесса, сопровождаемое нейтрализацией и повторным доказательством некоторых зависящих от них акторов.

Для изменения производных значений общих переменных некоторого процесса непосредственно в ходе доказательства актора P, принадлежащего этому процессу, используется (недетерминированный) встроенный предикат разрушающего присваивания

L := R.

Исполнение этого предиката осуществляется следующим образом:

1. Аргументы предиката унифицируются друг с другом.

2. После выполнения первого шага происходит согласование акторов процесса в соответствии с правилами, определёнными в разделе 7.3.

Доказательство предиката считается успешным в том и только в том случае, если завершаются успехом унификация его аргументов, а также последующее согласование акторов процесса.

Встроенный предикат разрушающего присваивания разрешается использовать с произвольным количеством аргументов:

':='(V_1,...,V_k).

Примечание. Встроенный предикат разрушающего присваивания является недетерминированным потому, что вызываемое им повторное доказательство акторов, в общем случае, может приводить к построению новых точек выбора.

Пример. Использование предиката разрушающего присваивания.

Рассмотрим поведение доказанного актора, определённого с помощью фрагмента программы

```
class 'Main':
x;
[
goal:–
     subgoal(x).
subgoal(1).
subgoal(3).
subgoal(5).
]
```

В результате исполнения предиката x:=5, актор goal будет нейтрализован и доказан повторно, производное значение общей переменной x станет равным 5.

Ссылки: актор 7.1, встроенный предикат 8, доказанный актор 7.1, доказательство актора 6.3.1, исполнение предиката 6.3.1, нейтрализация актора 7.1, общие переменные 7.2, повторные доказательства 7.1, принадлежать процессу 5.2, программа 4, производные значения 7.2, процесс 5.2, согласование акторов 7.3, унификация 3.3, **class** 2.1.2, goal 5.4.1.

8.2 Актуализация производных значений

Актуализацией производных значений общих переменных процесса G называется унификация локальных значений общих переменных (всех) активных акторов процесса G с соответствующими им производными значениями этого процесса.

Встроенный управляющий оператор

$$\mathrm{copy}(V_1,\ldots,V_k)$$

проверяет возможность актуализации производных значений общих переменных процесса. Если актуализация производных значений возможна, результатом его исполнения становится унификация переменных в составе аргументов $V_1,\ldots,V_k$ с соответствующими производными значениями, в противном случае исполнение оператора заканчивается неудачей.

Примечание. Обозначение оператора copy с помощью квадратных скобок рассмотрено в разделе 6.2.

Примечание. Использование оператора copy, в общем случае, может нарушить полноту программы относительно её декларативной семантики.

Пример. Использование оператора copy.

Предположим, что некоторому процессу G принадлежит доказанный актор goal, которому соответствует локальное значение общей переменной x=100.

```
goal:–
     subgoal_a(x).
subgoal_a(100).
subgoal_b:–
     [x],
     write("Shared Data = ",x).
```

В результате исполнения предиката subgoal_b процесса G, будет напечатано:

```
Shared Data = 100
```

Ссылки: активные акторы 7.1, актор 7.1, встроенный оператор 8, доказанный актор 7.1, исполнение предиката 6.3.1, локальные значения 7.2, общие переменные 7.2, переменная 2.1.1, принадлежать процессу 5.2, программа 4, производные значения 7.2, процесс 5.2, унификация 3.3, goal 5.4.1.

Приложение A

Сводка синтаксиса

(Данное приложение не является частью определения языка.)

A.1 Синтаксические правила языка

1.
letter = capital_letter | small_letter
capital_letter =
"A" | "B" | "C" | "D" | "E" | "F" | "G" |
"H" | "I" | "J" | "K" | "L" | "M" | "N" |
"O" | "P" | "Q" | "R" | "S" | "T" | "U" |
"V" | "W" | "X" | "Y" | "Z"
small_letter =
"a" | "b" | "c" | "d" | "e" | "f" | "g" |
"h" | "i" | "j" | "k" | "l" | "m" | "n" |
"o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w" | "x" | "y" | "z"
digit =
"0" | "1" | "2" | "3" | "4" | "5" | "6" |
"7" | "8" | "9"
letters_and_digits =
[letters_and_digits ["_"]] letter_or_digit
letter_or_digit = letter | digit

2.1.1.
variable =
    capital_letter [["_"] letters_and_digits] |
    "_" [letters_and_digits]

2.1.2.
symbol = simple_symbol | symbol_in_apostrophes
simple_symbol =
    small_letter [["_"] letters_and_digits]
symbol_in_apostrophes = ' { grapheme } '

2.1.3.
numerical_literal =
    extended_number [exponent] |
    digits "#" extended_number "#" [exponent] |
    ` grapheme
extended_number =
    letters_and_digits ["." letters_and_digits]
digits = [digits ["_"]] digit
exponent = e ["+" | "−"] digits
e = "E" | "e"

2.1.4.
segment_of_string = `"' { grapheme | "\" code } `"'
code = "b" | "t" | "n" | "v" | "f" | "r" | numerical_literal

3.
term =
    simple_term |
    compound_term |
    call_of_function_in_clause
functor = symbol | metavariable
metavariable = variable

3.1.
simple_term = constant | parameter
constant =
    symbol_in_apostrophes |
    ["−"] numerical_literal |
    string_literal |

“#” |
metavariable
string_literal = [string_literal] segment_of_string
parameter = variable | attribute

3.2.
compound_term =
structure | list | underdetermined_set

3.2.1.
structure = functor “(” terms_and_expressions “)”
terms_and_expressions =
[terms_and_expressions “,”] term_or_expression
term_or_expression = term | expression

3.2.2.
list = “[” [terms_and_expressions [“|” tail]] “]”
tail = parameter | call_of_function_in_clause | expression

3.2.3.
elements_of_set =
[elements_of_set “,”] element_of_set
element_of_set =
name_of_element [“:” term_or_expression] | attribute
name_of_element = symbol | numerical_literal
underdetermined_set =
[simple_term] “{” elements_and_tail_of_set “}”
elements_and_tail_of_set = [elements_of_set] [“|” tail]

4.
program = { declaration_of_class | declaration_of_project }

4.1.
declaration_of_class =
“**class**” heading_of_class “:” attributes “[” clauses “]”
heading_of_class =
name_of_class [“**specializing**” name_of_class]
name_of_class = symbol_in_apostrophes

4.1.1.
attributes = { definition_of_attribute “;” }

definition_of_attribute =
    [declarator_of_port ":"] attribute ["=" initializer]
declarator_of_port = "**suspending**" | "**protecting**"
attribute = simple_symbol

4.1.2.
initializer = term | constructor

4.1.3.
constructor = constructor_of_world | constructor_of_resident
constructor_of_world =
    simple_constructor | constructor_of_process
simple_constructor =
    "(" name_of_class { "," definition_of_attribute } ")"
constructor_of_process = "(" simple_constructor ")"
constructor_of_resident =
    [parameter_or_constructor] "??" simple_atom
parameter_or_constructor =
    target_parameter | constructor_of_world
target_parameter = parameter

4.2.
declaration_of_project =
    "**project**" ":" constructor_of_process

4.3.
package = [heading_of_package] import_commands program
heading_of_package = "**package**" name_of_package ":"
name_of_package = string_literal
import_commands = { import_command }
import_command =
    "**import**" imported_name "**from**" name_of_package ";"
imported_name = name_of_class ["**as**" name_of_class]

6.
clause = atom [":–" conjunction] "."
conjunction = [conjunction ","] subgoal
clauses = { clause }

6.1.
atom =
    simple_atom |

```
      binary_relation |
      declaration_of_function

6.1.1.
simple_atom =
      functor [ "(" [ terms_and_expressions [ "*" ] ] ")" ] |
      underdetermined_set |
      metavariable

6.1.2.
binary_relation =
      term_or_expression relational_operator term_or_expression
relational_operator =
      "==" | ":=" | "<" | ">" | "<>" | "<=" | ">="

6.1.3.
declaration_of_function =   simple_atom "=" term_or_expression

6.2.
subgoal =
      simple_subgoal |
      binary_relation |
      "[" [ terms_and_expressions ] "]" |
      "!"
simple_subgoal =
      [ [ target_parameter ] subgoal_infix ] simple_atom
subgoal_infix =   "?" | "<<" | "<-"

6.2.1.
call_of_function_in_clause =
      [ target_parameter ] "?" simple_atom |
      target_parameter "[" terms_and_expressions "]"

6.2.2.
expression =
      [ expression adding_operator ] addend |
      expression adding_operator term
addend =
      [ addend multiplying_operator ] multiplicand |
      addend multiplying_operator term
multiplicand =   [ "-" ] "(" expression ")"
```

adding_operator = “+” | “−”
multiplying_operator = “∗” | “/”

А.2 Дополнительные условия

2.

Текст программы рассматривается как последовательность лексем и разделителей.

Разделителями являются комментарии, а также пробелы и управляющие символы, не входящие в состав лексем и комментариев.

Сканирование текста всегда осуществляется слева направо.

В состав каждой лексемы включается по возможности большее число графических символов.

Фрагмент текста «:−» не является лексемой, если он расположен между лексемами «{» и «}», составляющими пару «открывающая скобка — закрывающая скобка».

Фрагмент текста «<−» не является лексемой, если он расположен непосредственно перед числовым литералом или ограничителем «(».

2.1.2.

Если символ не заключён в апострофы, его значение не может совпадать с ключевыми словами языка.

2.1.3.

По умолчанию основание числового литерала равно 10.

Значение каждой (расширенной) цифры числового литерала с основанием должно быть меньше основания.

В определении числового литерала не допускается (считается синтаксической ошибкой) использование пробела и управляющих символов.

Порядок целых числовых литералов не может содержать знак минус.

2.1.4.

В качестве кода в сегменте строки не допускается (считается синтаксической ошибкой) использование вещественных числовых литералов, а также числовых литералов, значения которых лежат за пределами некоторого интервала, определяемого конкретной реализацией языка.

3.

Метапеременные разрешается использовать в качестве функторов только в составе предложений и только при условии, что такой же метафунктор является именем предиката в заголовке рассматриваемого предложения.

Функтор, используемый в составе определения класса и совпадающий с некоторым атрибутом этого класса, должен быть символом в апострофах.

3.1.

В целых и вещественных числах с явно указанным основанием не разрешается использовать знак минус.

Метапеременные разрешается использовать в качестве простых термов только в составе предложений и только при условии, что такая же метапеременная является именем предиката или атомарной формулой в заголовке рассматриваемого предложения.

Значения целых чисел не могут выходить за пределы допустимого диапазона.

Значения вещественных чисел не могут выходить за пределы допустимого диапазона.

Значения числовых литералов с явно указанным основанием (выходящие за пределы допустимого диапазона) разрешается использовать в качестве битового представления отрицательных чисел.

Длина значений строковых литералов не может превышать максимальное допустимое значение.

3.2.3.

В качестве имён элементов недоопределённых множеств используются символы и неотрицательные целые числа.

Если в составе элемента недоопределённого множества не заданы терм или выражение после имени элемента, символ, используемый в качестве имени элемента, должен быть в апострофах.

Если недоопределённое множество используется в составе определения класса, то имена элементов множества, совпадающие с атрибутами этого класса, должны быть символами в апострофах.

Заголовок недоопределённого множества определяет элемент множества с именем 0 (ноль).

Если недоопределённое множество используется в составе определения атрибутов класса, в качестве хвоста этого множества разрешается использовать только переменные.

Недоопределённое множество не может содержать пары с одинаковыми именами элементов.

4.

Программа состоит из множества классов и целевого утверждения («проекта»).

В программе должны быть определены все классы, используемые проектом.

4.1.

В иерархии наследования классов, используемых проектом, запрещены циклические зависимости.

4.1.1.

Каждый атрибут должен быть объявлен во всех классах, связанных отношением наследования, в которых используется соответствующий слот.

В определении атрибутов класса не допускается однократное использование переменных, отличных от «_».

Не допускается повторное определение атрибутов класса (в том числе переопределение атрибута self).

4.1.3.

Отсутствие инициализатора в определении некоторого атрибута конструктора с именем Name является допустимым только в том случае, если рассматриваемый конструктор находится в области действия слота с именем Name.

В конструкторе экземпляра класса не допускается определение нескольких атрибутов с одинаковыми именами.

Не допускается использование символа self в качестве имени аргумента конструктора.

В качестве простых атомов в конструкторе резидента не разрешается использовать переменные.

4.2.

В определении проекта не допускается однократное использование переменных, отличных от «_».

4.3.

Несоответствие имени транслируемого пакета, не позволяющее однозначно сопоставить его с именем исходного файла, является синтаксической ошибкой.

Последовательность команд импорта, которая делает видимыми некоторые импортируемые классы под одним и тем же именем, является синтаксической ошибкой.

Использование в пакете нескольких команд импорта, которые делают видимым один и тот же класс из одного и того же пакета, является синтаксической ошибкой.

4.4.

Повторное определение класса в пакете, а также повторное определение класса в наборе совместно транслируемых пакетов являются синтаксической ошибкой.

Повторное определение проекта в пакете, а также повторное определение проекта в наборе совместно транслируемых пакетов являются синтаксической ошибкой.

6.

В составе предложения не допускается однократное использование переменных, отличных от «_».

Предложения, не являющиеся метапредложениями, должны принадлежать одной группе («процедуре»), если совпадают имена и арность предикатных символов заголовков этих предложений.

В определении класса следует группировать процедуры с одинаковыми именами предикатных символов заголовков входящих в них предложений.

Метапредложения, в заголовке которых присутствует предикатный символ, и в качестве этого предикатного символа задан символ, должны быть сгруппированы с предложениями с таким же именем предикатного символа заголовков.

6.1.

Последний аргумент атомарной формулы может быть помечен «*» только в том случае, если он является переменной.

Переменная в составе атомарной формулы подцели предложения может быть помечена «*» лишь в том случае, если она таким же образом помечена в заголовке предложения и не является анонимной переменной «_».

Для обозначения списка аргументов предиката переменной арности не разрешается использовать метафункторы.

6.1.3.

В качестве простых атомов в составе объявлений функций не разрешается использовать метапеременные.

6.2.

Если инфикс подцели равен «<<» или «<−», в качестве простого атома этой подцели не разрешается использовать метапеременные.

6.2.1.

В качестве простых атомов в составе вызовов функций не разрешается использовать метапеременные.

8.

Встроенными предикатами языка являются goal(), alarm(E), ''(Set) и element(Value,I_1,...,I_k), определяемые в тексте программы, а также предопределённые предикаты '=='(V_1,...,V_k), ':='(V_1,...,V_k), true[(...)], fail.

Встроенными управляющими операторами языка являются операторы copy(V_1,...,V_k), '!', break[(E)], spypoint(...).

В программе не допускается определение предикатов, имена которых совпадают с именами предопределённых предикатов и встроенных управляющих операторов.

Не разрешается использование имён предопределённых предикатов и встроенных управляющих операторов в качестве предикатных символов в акторных и дальних вызовах.

Неверное число аргументов в предопределённых предикатах и встроенных управляющих операторах является синтаксической ошибкой.

А.3 Перекрёстные ссылки синтаксиса

Приложение B

Свойства, зависящие от реализации

(Данное приложение не является частью определения языка.)

2.1.3.

Максимальная относительная погрешность вещественных чисел.

Кодировка графических символов, используемая в числовых литералах вида ‘ графема.

2.1.4.

Диапазон допустимых значений числовых литералов, используемых в качестве кодов в строковых литералах.

3.1.

Диапазон допустимых целых чисел.

Диапазон допустимых вещественных чисел.

Битовое представление отрицательных чисел.

Максимальная допустимая длина значений строковых литералов.

4.3.

Имя и местонахождение системного каталога.

4.4.

Поддержка совместной, раздельной трансляции исходных файлов, или обоих способов трансляции.

Поддержка автоматической трансляции пакетов, указанных в командах импорта в составе других пакетов.

Автоматическая трансляция пакетов, указанных в командах импорта, может осуществляться отдельно от других пакетов или как совместная трансляция этих пакетов.

Структура программной библиотеки.

Максимальный допустимый размер исходного файла.

5.3.

Правила упорядочения термов (значений функций).

7.5.

Действия встроенного обработчика исключительных ситуаций.

Обозначения исключительных ситуаций, вызываемых на системном уровне (за пределами текста программы).

8.

Результаты исполнения встроенного управляющего оператора «обращение к отладчику».

Приложение C

Термины и определения

(Данное приложение не является частью определения языка.)

Активизация актора (activation of actor) — перевод актора в активное состояние.

Активизация порта (activation of port) — операции, осуществляемые в начале обработки потокового сообщения, пришедшего в процесс через порт: 1) Установка локального значения актора-представителя порта равным значению порта на момент начала рассматриваемой фазы исполнения процесса. 2) Объявление актора-представителя порта активным (считается, что доказательство этого актора успешно завершено).

Активный актор (active actor) — одно из возможных состояний актора — актор, доказательство которого происходит (произошло) в ходе текущей фазы исполнения процесса. Актор считается активным с момента начала его доказательства до момента завершения фазы.

Активный процесс (active process) — процесс, находящийся на очередной фазе своего исполнения.

Актор (actor) — подцель доказательства, соответствующая некоторому акторному вызову предиката.

Акторный вызов предиката (actor call of predicate) — тип вызова предиката, используемый для определения акторов.

Акторный механизм (actor mechanism) — стратегия управления Акторного Пролога. Акторный механизм является расширением стандартной стратегии управления («поиск слева направо в глубину с возвратом»). Акторный механизм допускает разрушающее присваивание значений переменным, обеспечивая корректность доказательства теоремы

с помощью повторного доказательства отдельных акторов.

Актор-представитель (actor representative) — вспомогательный актор, поставленный в соответствие порту процесса. Акторы-представители портов процесса предназначены для хранения информации, приходящей в процесс через порты в виде потоковых сообщений.

Актуализация производных значений (actualization of derived values) — унификация локальных значений общих переменных (всех) активных акторов некоторого процесса с соответствующими им производными значениями этого процесса.

Актуальное значение переменной (actual value of variable) — значение общей переменной некоторого процесса, которое можно получить, унифицировав её локальные значения, соответствующие (всем) активным акторам этого процесса. Считается, что общие переменные процесса, не соответствующие ни одному из его активных акторов, не имеют актуальных значений. В ходе исполнения программы локальные значения общих переменных каждого активного актора автоматически поддерживаются равными актуальным значениям.

Анонимная переменная (anonymous variable) — переменная «_». Считается, что все анонимные переменные являются некоторыми уникальными, однократно использованными именами.

Аргумент (argument) — 1. Терм или выражение, заключённые в скобки в составе предиката или составного терма. 2. В составе конструктора экземпляра класса: пара «атрибут= инициализатор», которой может предшествовать описатель порта.

Арность (arity) — количество аргументов.

Атомарная формула, атом (atomic formula, atom) — предикат или метапеременная, используемые в качестве элементарного объекта, обладающего значением истинности.

Атрибут (attribute) — имя слота; объявляется во всех классах, в которых используется соответствующий слот. Атрибут self — предопределённый, он обозначает непосредственно тот мир, в котором это имя используется.

Библиотечный модуль (library unit) — элементарный программный модуль (определение класса или определение проекта), записанный в программную библиотеку в результате трансляции.

Бинарное отношение (binary relation) — атомарная формула, состоящая из двух операндов, соединённых оператором отношения.

Ближний вызов предиката (near call of predicate) — вызов предиката, в котором не указан явно мир, где этот вызов должен быть исполнен.

Ближний вызов исполняется в том же самом мире, в котором исполняется предложение, содержащее данный вызов.

Владелец резидента (owner of resident) — то же что «создатель резидента». Резидент передаёт результаты своей работы владельцу с помощью потоковых сообщений.

Вложенные акторы (nested actors) — Актор Q называется «вложенным» по отношению к актору P, если эти акторы принадлежат одному процессу, и доказательство актора Q, результаты которого в данный момент не отменены, происходит (произошло) в ходе доказательства актора P.

Вложенные миры (nested worlds) — Мир B называется вложенным по отношению к миру A, если конструктор мира B является инициализатором слота мира A или какого-либо мира E, вложенного по отношению к A.

Встроенный предикат (built-in predicate) — предикат, являющийся составной частью определения языка. Встроенными предикатами называются (являются) предикаты goal(), alarm(E), ''(S), element(V,I_1,...,I_k), а также все предопределённые предикаты.

Встроенный управляющий оператор (built-in control statement) — управляющий оператор, являющийся составной частью определения языка. Встроенными называются управляющие операторы copy(V_1,...,V_k), '!', break[(E)], spypoint(...).

Вызов предиката (predicate call) — синтаксическая конструкция, определяющая экземпляр класса, в котором этот вызов должен быть исполнен, тип вызова (ближний или дальний), а также атомарную формулу вызова.

Вызов функции (call of function) — синтаксическая конструкция, имитирующая вызов подпрограммы-функции, возвращающей некоторое значение — терм. Вызовами функций являются атомарные формулы, заданные в конструкторах резидентов, а также специальные синтаксические конструкции в составе предложений.

Выражение (expression) — синтаксическая конструкция, представляющая собой видоизменённый вызов функции.

Глобальное значение переменной (global value of variable) — некоторый терм, поставленный в соответствие общей переменной, используемой для передачи потоковых сообщений между процессами, единый для всех этих процессов. Текущему глобальному значению переменной соответствуют сорт и производитель.

Глобальные операции (global operations) — операции, в которых используются локальные значения, принадлежащие активным и

доказанным акторам некоторого процесса — сопоставление локальных значений общих переменных, актуализация производных значений общих переменных, передача потоковых сообщений из процесса.

Графический символ, графема (graphic character, grapheme) — элемент набора символов ASCII, имеющий визуальное представление в виде отпечатанного знака или пробела.

Дальний вызов предиката (far call of predicate) — вызов предиката, в котором явным образом (с помощью переменной или атрибута) указан мир, в котором этот вызов должен быть исполнен.

Детерминированный (deterministic) — такой, в результате исполнения которого не возникают новые точки выбора.

Доказанный актор (proven actor) — одно из возможных состояний актора — актор, доказательство которого завершилось успехом на одной из фаз F исполнения процесса G и в данный момент не отменено. Актор считается доказанным с момента (успешного) окончания фазы F процесса G до (возможной) нейтрализации рассматриваемого актора на одной из последующих фаз процесса G.

Доказанный процесс (proven process) — одно из возможных состояний процесса — характеризуется тем, что все акторы, принадлежащие процессу, согласованы.

Доказательство — 1. (proving) исполнение чего-либо (например, доказательство процесса, актора, конструктора, подцели доказательства, подцели предложения, предиката, целевого утверждения); проверка чего-либо (например, доказательство существования производных значений переменных); 2. (proof) результаты успешного доказательства чего-либо.

Доказательство актора — 1. (proving of actor) исполнение (первое или повторное) акторного вызова предиката; 2. (proof of actor) результаты успешного исполнения актора (в том числе, возможно, точки выбора и значения переменных), включая результаты доказательства вложенных по отношению к нему акторов.

Заголовок недоопределённого множества (heading of underdetermined set) — значение элемента с именем 0 (ноль). Заголовок может быть задан перед фигурными скобками недоопределённого множества.

Заголовок предложения (heading of clause) — атомарная формула в начале предложения (стоящая перед ограничителем «:–» или «.»).

Задержанная подцель (suspended subgoal) — подцель доказатель-

ства, исполнение которой было отложено механизмом задержки исполнения подцелей.

Задерживающее значение (suspending value) — значение потокового сообщения, переменной или порта, равное спейсеру # или несвязанной переменной (в том числе, пустому значению).

Защищающий порт (protecting port) — разновидность портов процессов, обладающая следующими свойствами: 1) Все потоковые сообщения, передаваемые процессом через защищающий порт автоматически объявляются защищёнными. 2) Значения всех незащищённых сообщений, принимаемых процессом через защищающий порт игнорируются в ходе обработки этих сообщений.

Защищённое сообщение (protected message) — разновидность потоковых сообщений, используемая для управления передачей потоковых сообщений между процессами. В ходе передачи потоковых сообщений из некоторого процесса G незащищённое сообщение не может изменить глобальное значение, переданное в виде защищённого сообщения из какого-либо другого процесса (отличного от G).

Знак операции (operator) — ограничитель, специальным образом используемый в составе выражений и атомарных формул.

Значение лексемы (value of token) — наименование смысловых единиц, создаваемых лексическим анализатором и соответствующих лексемам, обнаруженным в тексте программы. Последовательность значений лексем передаётся синтаксическому анализатору в качестве результатов работы лексического анализатора.

Значение переменной (value of variable) — 1. Значением лексемы «переменная» является соответствующая ей последовательность графем. 2. Значением терма «переменная» является значение лексемы «переменная», до тех пор пока переменная (терм) не будет связана с какой-либо константой, составным термом или миром. Значением связанной переменной является соответствующий элемент данных, мир или спейсер.

Значение порта (value of port) — некоторое значение терма.

Значение потокового сообщения (value of flow message) — некоторое значение терма.

Значение слота (value of slot) — некоторое значение терма.

Значение терма (value of term) — элемент данных, мир, спейсер или, если терм является несвязанной переменной, значение лексемы «переменная».

Иерархия наследования (inheritance hierarchy) — отношение наследования, задаваемое на множестве классов программы. В Акторном Прологе используется одиночное наследование, согласно которому у класса может быть не более одного непосредственного предка.

Импорт класса (import of class) — обеспечение видимости имени класса за пределами пакета, в котором этот класс определён. Импорт классов осуществляется с помощью специальной команды импорта.

Импортируемый класс (imported class) — класс, заданный в команде импорта.

Инициализатор слота (initializer of slot) — синтаксическая конструкция, определяющая начальное значение слота.

Инициализация процесса (initialization of process) — фаза исполнения процесса, перед началом которой он находился в состоянии «сформированный».

Инициализирующее сообщение (initializing message) — специальная разновидность переключающих потоковых сообщений. Инициализирующее сообщение автоматически посылается процессу в ходе его формирования, независимо от того, имеет ли он какие-либо порты, а также от состояния этих портов.

Интерференция потоковых сообщений (interference of flow messages) — одновременная обработка процессом нескольких потоковых сообщений.

Информационные сообщения (informational messages) — разновидность сообщений, характеризуемая тем, что: 1) Обработка таких сообщений процессом откладывается до тех пор, пока он не окажется в состоянии «доказанный». 2) После их обработки процесс всегда оказывается в состоянии «доказанный».

Исключительная ситуация (exception) — аварийное состояние вычислительного процесса, зарегистрированное во время исполнения программы.

Исполнение вызова предиката (execution of predicate call) — то же что «исполнение предиката» (execution of predicate). Общая схема исполнения предиката включает: 1) Выбор предложения с подходящим заголовком. 2) Построение точки выбора (если следом за выбранным предложением в рассматриваемом мире расположены ещё не исследованные предложения). 3) Построение копии выбранного предложения с переименованием переменных и подстановкой значений слотов. 4) Исполнение построенного предложения.

Исполнение конструктора (execution of constructor) — последовательность действий, осуществляемая для создания нового экземпляра класса, процесса, резидента.

Исполнение предложения (execution of clause) — Исполнение предложения включает: 1) Унификацию функтора и аргументов исполняемого вызова предиката с функтором и аргументами заголовка предложения или унификацию вызова предиката с метапеременной (если заголовком предложения является метапеременная). 2) Пересмотр списка задержанных подцелей. 3) Исполнение соответствующих подцелей доказательства.

Исполнение программы (execution of program) — Исполнением программы называется построение и согласование процессов. Исполнение программы начинается с доказательства конструктора процесса, заданного в определении проекта, а также формирования процесса, построенного в результате доказательства этого конструктора.

Исполнение процесса (execution of process) — доказательство акторов, принадлежащих процессу.

Использование класса (use of class) — класс C (или проект) «использует» класс E, если E является предком C в иерархии наследования классов, а также если C (проект) или кто-либо из его предков содержит конструктор экземпляра класса E (или конструктор экземпляра класса F, такого что класс F использует класс E), не считая тех конструкторов, которые входят в состав инициализаторов, перекрываемых во время построения соответствующих миров.

Использование переменной актором (use of variable by actor) — Будем говорить, что актор P «использует» переменную V (или что переменная V «соответствует», «принадлежит» актору P), если переменная V входит в состав подцели доказательства P или какой-либо другой подцели доказательства Q, построенной в ходе: а) текущего доказательства подцели P, если исполнение актора ещё не закончено, б) последнего завершившегося успехом доказательства актора P, если исполнение актора закончено, и он не является нейтральным — не считая тех подцелей доказательства Q, которые были построены в ходе доказательства акторов, вложенных по отношению к P.

Используемый процесс (used process) — обобщающее название для состояний процесса «сформированный», «доказанный», «неудачный». Это понятие противоположно понятию «неиспользуемый процесс».

Исходный файл (source file) — файл, содержащий текст (фрагмент

текста) программы, который может быть оттранслирован независимо от остальных. В Акторном Прологе каждый исходный файл считается пакетом, и каждый пакет должен храниться в отдельном исходном файле.

Класс (class) — набор предложений языка, имеющий уникальное имя и входящий в состав иерархии наследования.

Ключевое слово (keyword) — символическое обозначение, по которому транслятор распознаёт некоторые синтаксические конструкции.

Комментарий (comment) — часть текста программы, не влияющая на её исполнение и служащая для документирования и облегчения чтения программы человеком.

Константа (constant) — простой терм, отличный от переменной и атрибута.

Конструктор (constructor) — утверждение о существовании экземпляра класса или резидента; в результате доказательства конструкторов происходит построение новых экземпляров классов и резидентов. Различаются конструкторы миров (а именно простые конструкторы и конструкторы процессов) и конструкторы резидентов.

Конструктор мира (constructor of world) — простой конструктор или конструктор процесса.

Конструктор процесса (constructor of process) — разновидность конструктора — утверждение о существовании процесса. Доказательство конструктора процесса приводит к созданию нового процесса.

Конструктор резидента (constructor of resident) — разновидность конструктора — утверждение о существовании резидента. Доказательство конструктора резидента приводит к созданию нового резидента.

Лексема (token) — наименование смысловых единиц, распознаваемых лексическим анализатором в тексте программы. Лексемами являются: переменные, символы и ключевые слова, целые числовые литералы, вещественные числовые литералы, сегменты строк, ограничители.

Локальное значение общей переменной (local value of common variable) — значение общей переменной, соответствующее некоторому конкретному актору. В Акторном Прологе каждый актор хранит свои собственные (локальные) значения общих переменных.

Максимальная относительная погрешность (maximal relative error) — ошибка представления вещественных чисел; определяет максимальное количество значащих цифр вещественного числа, воспринимаемое транслятором.

Метаатом (metaatom) — то же что «метапредикат».

Метапеременная (metavariable) — переменная, обозначающая предикат (в этом случае метапеременная является метапредикатом), функтор (в этом случае она является метафунктором) или список аргументов предиката с переменным числом аргументов.

Метапредикат (metapredicate) — предикат переменной арности, метапеременная, используемая в качестве атома или атом, в качестве функтора которого используется метапеременная.

Метапредложение (metaclause) — предложение, в котором используются метапеременные.

Метафунктор (metafunctor) — переменная, используемая в качестве функтора.

Механизм задержки исполнения (mechanism of suspension of execution) — вспомогательная стратегия управления, откладывающая исполнение выделенных подцелей до тех пор, пока не будет вычислена некоторая информация, необходимая для корректного исполнения этих подцелей.

Мир (world) — то же что «экземпляр класса».

Наследование (inheritance) — принцип формализации знаний, в соответствии с которым в Акторном Прологе набор предложений экземпляра некоторого класса C включает предложения класса C, а также предложения всех классов, являющихся предками C в иерархии наследования, заданной на множестве классов программы.

Начальное значение слота (initial value of slot) — значение слота, созданное во время его построения.

Недетерминированный (non-deterministic) — имеющий несколько возможных путей исполнения, которые могут быть перечислены с помощью отката.

Недоопределённое множество (underdetermined set) — составной терм, построенный из набора (возможно, пустого) элементов, заключённого в фигурные скобки. Элементы недоопределённого множества задаются в виде пар «имя_элемента: терм_или_выражение». В случае если набор элементов множества не является пустым, в состав множества может быть включён дополнительный компонент, обозначающий неопределённый остаток (хвост) множества.

Незащищённое сообщение (unprotected message) — потоковое сообщение, не являющееся защищённым.

Неиспользуемый процесс (unused process) — одно из возможных состояний процесса — характеризуется тем, что на некоторые отключающие

порты процесса поданы задерживающие значения. Неиспользуемый процесс не принимает и не посылает никакие сообщения. Считается, что неиспользуемый процесс не имеет никаких производных значений, и все его акторы согласованы.

Нейтрализация актора (neutralization of actor) — отмена всех результатов доказательства актора, за исключением результатов доказательства вложенных по отношению к нему акторов.

Нейтрализация процесса (neutralization of process) — переход процесса в состояние «неудачный» (в случае неудачного или аварийного завершения обработки некоторого переключающего сообщения).

Нейтральный актор (neutral actor) — актор, предыдущее доказательство которого отменено, а повторное доказательство ещё не началось.

Непустое значение сообщения (non-empty value of message) — значение непустого потокового сообщения.

Непустое потоковое сообщение (non-empty flow message) — потоковое сообщение, не являющееся пустым.

Несвязанная переменная (unbound variable) — переменная, не связанная с константой, составным термом или миром.

Несогласованный порт (inconsistent port) — одно из двух возможных состояний порта процесса — порт становится «несогласованным», когда процесс получает через него потоковое сообщение.

Неудачный процесс (failed process) — одно из возможных состояний процесса — характеризуется тем, что акторы процесса выведены из согласованного состояния. В этом состоянии приостанавливается обработка любых информационных сообщений, принимаемых процессом.

Обозначение исключительной ситуации (designation of exception) — символ или неотрицательное целое число.

Обработка исключительной ситуации (processing of exception) — действия, осуществляемые программой в случае возникновения исключительной ситуации.

Обработка сообщения (processing of message) — фаза исполнения процесса — действия, осуществляемые процессом в случае получения сообщения.

Общая переменная (common variable) — переменная, которая используется (или может быть использована) несколькими акторами.

Объявление функции (declaration of function) — разновидность атомарной формулы — синтаксическая конструкция, имитирующая

заголовок подпрограммы-функции, возвращающей некоторое значение — терм. В результате трансляции объявления функций преобразуются в предикаты.

Объявленный процесс (declared process) — одно из возможных состояний процесса — характеризуется тем, что соответствующие ему экземпляры классов ещё не сформированы.

Ограничитель (delimiter) — разновидность лексемы — последовательность из одного или нескольких специальных символов, используемая в синтаксических конструкциях языка.

Оператор отсечения, '!' (cut statement) — встроенный управляющий оператор — отсечение устраняет все неисследованные пути (точки выбора), которые встретились с момента начала исполнения предиката, в соответствие которому было поставлено предложение, содержащее оператор.

Описатель порта (declarator of port) — ключевое слово, с помощью которого задаётся сорт порта. Описателями портов служат ключевые слова «suspending» и «protecting», обозначающие «отключающий» и «защищающий» соответственно.

Освобождение общих переменных процессом (releasing common variables by process) — передача пустых сообщений из процесса; осуществляется каждый раз при переходе этого процесса из состояния «доказанный» в состояние «неудачный» или «неиспользуемый».

Откат (backtracking) — возобновление исполнения процесса, начиная с последней (неустранённой оператором отсечения) точки выбора. В результате отката осуществляется восстановление состояний всех акторов рассматриваемого процесса на момент прохождения упомянутой точки выбора (в том числе отмена всех связываний и сцеплений переменных, произошедших в акторах с момента прохождения этой точки, отмена нейтрализации и результатов повторных доказательств акторов, а также отмена всех изменений, внесённых в список задержанных подцелей).

Отключающее значение (suspending value) — то же, что «задерживающее значение».

Отключающее сообщение (suspending flow message) — потоковое сообщение, значение которого является задерживающим. Поступление отключающего потокового сообщения на любой отключающий порт процесса вызывает отключение этого процесса.

Отключающий порт (suspending port) — разновидность портов процессов, обладающая следующими свойствами: 1) При получении через отключающий порт задерживающего значения процесс автоматически

переводится в состояние «неиспользуемый». Когда значения всех отключающих портов процесса перестают быть задерживающими, он автоматически возвращается в состояние «используемый». 2) Отключающий порт R процесса G всегда активизируется в начале фазы исполнения процесса G, если производителем текущего значения порта R является процесс, отличный от G.

Отключение процесса (disconnection of process) — перевод процесса в состояние «неиспользуемый».

Пакет (package) — совокупность классов, связанных между собой по смыслу. В Акторном Прологе каждый пакет должен храниться в отдельном исходном файле, и каждый исходный файл программы считается отдельным пакетом. Каждый пакет обладает собственной областью видимости имён классов: имена классов, используемые внутри пакета, не видны из других пакетов до тех пор, пока не будут в них импортированы.

Параметр (parameter) — переменная или атрибут.

Передача сообщения (message passing) — последовательность действий, реализующая распространение информации из одного процесса в другие. В общем случае, передача сообщений из некоторого процесса может осуществляться каждый раз после завершения очередной фазы его исполнения.

Переключающие сообщения (switching messages) — разновидность сообщений, характеризуемая тем, что: 1) В результате обработки такого сообщения процесс может перейти в состояние «доказанный», «неудачный» или (остаться в состоянии) «сформированный». 2) Обработка переключающих сообщений осуществляется процессом независимо от того, в каком состоянии — «доказанный», «неудачный» или «сформированный» — он перед этим находился.

Перекрытие инициализаторов (overriding of initializers) — осуществляется в ходе построения слотов экземпляра класса во время исполнения простого конструктора экземпляра класса — замена инициализаторов слотов в определении атрибутов класса другими инициализаторами, заданными в конструкторе или в классах, являющихся потомками рассматриваемого.

Перекрытие описателей портов (overriding of declarators of ports) — осуществляется в ходе построения слотов экземпляра класса во время исполнения простого конструктора экземпляра класса — замена описателей портов в определении атрибутов класса другими описателями портов, заданными в конструкторе или в классах, являющихся потомками

рассматриваемого. Перекрытие описателей портов происходит независимо от перекрытия инициализаторов слотов.

Переменная (variable) — разновидность лексемы — имя, начинающееся с большой буквы или символа подчёркивания.

Пересмотр списка задержанных подцелей (revision of list of suspended values) — один из этапов исполнения предложения — включает следующие действия: 1) Элементы списка задержанных подцелей просматриваются в том порядке, в котором они были в него добавлены. 2) При обнаружении каждого элемента списка, значение целевого параметра которого не является задерживающим, найденная подцель исключается из рассматриваемого списка и исполняется.

Повторное доказательство актора (repeated proving of actor) — повторение доказательства актора с самого начала.

Поглощение сообщения (absorption of message) — отмена сообщения, произошедшая вследствие неудачного или аварийного завершения его обработки. В случае поглощения сообщения, состояние процесса, обрабатывающего это сообщение, восстанавливается на момент, предшествовавший обработке этого сообщения.

Подключение процесса (connection of process) — перевод процесса в состояние «используемый».

Подцель доказательства (subgoal of proving) — подцель копии предложения, построенной в ходе исполнения некоторого вызова предиката.

Подцель предложения (subgoal of clause) — составная часть предложения — вызов предиката, исполнение которого осуществляется в ходе исполнения предложения.

Порт (port) — переменная процесса, которая может принадлежать акторам других процессов. В ходе исполнения программы каждому порту процесса ставятся в соответствие сорт, состояние, актор-представитель, а также текущее значение, сорт текущего значения, производитель текущего значения.

Построение процесса (generation of process) — Построение процесса происходит в результате доказательства конструктора процесса, однако при этом доказательство конструктора процесса не приводит к построению пространства поиска созданного процесса. Построение соответствующего пространства поиска и слотов процесса осуществляется позже, в ходе формирования процесса.

Построение резидента (generation of resident) — Построение резидента происходит в результате доказательства конструктора резидента.

Построение резидента включает следующие действия: 1) Переменная, созданная в качестве начального значения слота процесса-владельца, инициализатором которого является конструктор резидента, объявляется защищающим портом резидента. 2) Все остальные общие переменные, заданные в составе конструктора резидента, объявляются простыми портами резидента. 3) Новый резидент начинает функционировать.

Построение слота (generation of slot) — операция создания слота. Одновременно с созданием каждого слота, если для этого слота задан инициализатор, создаётся начальное значение слота. Все слоты, не получившие значения в ходе своего создания, получают в качестве начальных значений уникальные общие переменные.

Построение экземпляра класса (generation of class instance) — создание нового экземпляра класса; включает формирование экземпляра класса, а также доказательство предиката goal() во всех мирах, сформированных на этапе формирования экземпляра класса. В ходе построения экземпляра класса каждая автоматически исполняемая подцель goal объявляется новым актором.

Потоковое сообщение (flow message) — сообщение, реализующее передачу производных значений общих переменных из одного процесса в другие. Потоковые сообщения передаются и принимаются через порты процессов.

Правило (rule) — то же что «предложение».

Правило второго порядка (second order rule) — логическая формула, в которой переменными могут быть обозначены не только данные, но и предикаты. В Акторном Прологе правила второго порядка имитируются с помощью недоопределённых множеств и метапредложений.

Предикат (predicate) — синтаксическая конструкция, состоящая из предикатного символа (функтора) и последовательности (возможно, пустой) аргументов. Предикат может обозначать некоторую подпрограмму или набор подпрограмм (если речь идёт о предикате с переменным числом аргументов).

Предикатный символ (predicate symbol) — функтор, используемый в качестве имени предиката.

Предикат с переменным числом аргументов (variable arity predicate) — то же что «предикат переменной арности» — предикат, последним аргументом которого является переменная, помеченная ограничителем «$*$». В общем случае, предикат с переменным числом аргументов обозначает некоторый набор подпрограмм, имеющих

одинаковую структуру, но различную арность заголовков.

Предложение (clause) — одна из составных частей определения класса — логическое правило, состоящее из заголовка и последовательности (возможно, пустой) подцелей.

Предопределённый предикат (predefined predicate) — предикат, являющийся составной частью определения языка и обозначающий некоторую подпрограмму, не требующую определения в тексте программы, обладающую декларативной семантикой. Предопределёнными предикатами называются (являются) предикаты '=='(V_1,...,V_k), ':='(V_1,...,V_k), true[(...)], fail.

Принадлежать актору (belong to actor) — см. «использование переменной актором».

Принадлежать процессу (belong to process) — Считается, что некоторый актор «принадлежит» процессу G, если этот актор доказывается, доказан или должен быть доказан в мире, входящем в состав процесса G.

Проверка вхождения (occurence check) — проверка, осуществляемая в ходе унификации, предотвращающая (запрещающая) связывание переменной с составными термами, содержащими эту переменную. В соответствии с семантикой Акторного Пролога, проверка вхождения не распространяется на переменные в составе миров, являющихся компонентами унифицируемых термов.

Программа (program) — множество определений классов и определение проекта.

Программная библиотека (program library) — файл, в котором хранятся оттранслированные программные модули.

Программный модуль (program unit) — то же что «исходный файл».

Проект (project) — целевое утверждение программы — некоторый конструктор процесса.

Производитель глобального значения (producer of global value) — процесс, построивший текущее глобальное значение общей переменной.

Производитель значения порта (producer of value of port) — процесс, построивший текущее значение порта.

Производитель потокового сообщения (producer of flow message) — процесс, из которого передано потоковое сообщение.

Производное значение общей переменной процесса (derived value of common variable of process) — значение, которое можно получить (если оно существует), унифицировав локальные значения общей переменной, соответствующие всем активным и доказанным акторам

процесса. В случае если общая переменная процесса (например, некоторый порт процесса) не соответствует ни одному из активных или доказанных акторов, её производным значением считается анонимная переменная «_».

Простой атом (simple atom) — простейшая разновидность атомарной формулы — функтор с соответствующим количеством аргументов, недоопределённое множество или метапеременная.

Простой вызов предиката (simple call of predicate) — Простыми вызовами предикатов называются (являются) все вызовы, про которые не сказано, что они являются акторными.

Простой конструктор (simple constructor) — элементарное логическое утверждение о существовании экземпляра класса.

Простой порт (plain port) — порт процесса, который не является ни отключающим, ни защищающим.

Простой терм (simple term) — элементарная синтаксическая конструкция, обозначающая данные и миры. Простыми термами являются константы (символ, целое число, вещественное число, строковый литерал, спейсер #, метапеременная, обозначающая терм в метапредложении), а также параметры.

Процедура (procedure) — набор предложений класса (не являющихся метапредложениями) с одинаковыми предикатными символами атомарных формул заголовков (совпадают как имя, так и арность функторов).

Процесс (process) — экземпляр класса, предложения которого исполняются параллельно по отношению к предложениям других процессов. В языке используется только асинхронное взаимодействие между процессами, поэтому предикаты каждого процесса обладают декларативной семантикой, не зависящей от других процессов.

Прямое сообщение (direct message) — сообщение, реализующее исполнение дальнего вызова предиката из одного процесса в другом.

Пустое значение (empty value) — анонимная переменная «_» (используемая в качестве значения пустого сообщения).

Пустое сообщение (empty message) — специальная разновидность переключающих потоковых сообщений. Значение пустого сообщения равно анонимной переменной «_».

Разделитель (separator) — один или несколько графических и управляющих символов, разделяющих лексемы в тексте программы. Разделителями являются комментарии, а также пробелы и управляющие символы, не входящие в состав лексем и комментариев.

Разрушающее присваивание (destructive assignment) — В

Акторном Прологе разрушающим присваиванием называется изменение производных значений общих переменных некоторого процесса, сопровождаемое нейтрализацией и повторным доказательством некоторых зависящих от них акторов. Акторный механизм языка гарантирует логическую корректность программ, в которых используется разрушающее присваивание.

Расширенные цифры (extended digits) — цифры и буквы от «A» до «Z» (от «a» до «z»), используемые для определения числовых литералов с основанием.

Резидент (resident) — специальная активная сущность, отслеживающая состояния некоторых («целевых») процессов и передающая собранную информацию своему владельцу. Резиденты создаются в результате доказательства конструкторов резидентов.

Связывание переменной (binding of variable) — замена всех вхождений переменной некоторой константой, составным термом или экземпляром класса. В Акторном Прологе область действия операции связывания переменной всегда ограничена множеством вхождений, принадлежащих некоторым конкретным акторам.

Сегмент строки (string segment) — лексема, обозначающая цепочку графических и управляющих символов.

Символ (symbol) — разновидность лексемы — имя, начинающееся с маленькой буквы или заключённое в апострофы.

Системный каталог (system directory) — каталог в файловой системе компьютера, в котором по умолчанию хранятся пакеты. Системный каталог определяется реализацией языка.

Слот (slot) — составная часть экземпляра класса, характеризуемая именем и значением. Именем слота является некоторый атрибут, значением слота — терм.

Согласование акторов (coordination of actors) — действия, осуществляемые для обеспечения согласованности акторов некоторого процесса — попытка согласовать локальные значения общих переменных акторов процесса. Согласование акторов включает сопоставление локальных значений общих переменных акторов, а также повторное доказательство акторов, нейтрализованных в ходе проведённого сопоставления локальных значений.

Согласование процессов (coordination of processes) — действия, осуществляемые для обеспечения согласованности процессов. Согласование процессов происходит посредством обмена асинхронными сообщениями.

Согласованность акторов (consistency of actors) — Акторы

процесса считаются согласованными между собой, если: 1) Все акторы, принадлежащие процессу, хотя бы один раз были доказаны. 2) Существуют производные значения общих переменных этого процесса.

Согласованность процессов (consistency of processes) — Считается, что некоторые процессы «согласованы» между собой, если: 1) Все они находятся в состояниях «доказан» и «неиспользуемый». 2) Не требуется обработка потоковых и прямых сообщений процессами, находящимися в состоянии «доказан». 3) Не требуется обработка потоковых сообщений процессами, находящимися в состоянии «неиспользуемый». 4) Производные значения общих переменных всех процессов могут быть унифицированы.

Согласованный порт (consistent port) — одно из двух возможных состояний порта — порт может перейти в это состояние в ходе отправления или обработки потокового сообщения.

Создатель процесса (creator of process) — процесс, одному из слотов миров которого соответствовал инициализатор — конструктор рассматриваемого процесса.

Создатель резидента (creator of resident) — процесс, одному из слотов миров которого соответствовал инициализатор — конструктор резидента. Создатель резидента является его «владельцем».

Сообщение (message) — некоторое количество информации, передаваемое между процессами, представляющее для них единое целое. Различаются прямые и потоковые, а также переключающие и информационные сообщения. В языке используются потоковые переключающие, а также прямые информационные и прямые переключающие сообщения.

Соответствие переменной актору (correspondence between variable and some actor) — то же, что «использование переменной актором».

Сопоставление локальных значений (comparison of local values) — первый этап согласования акторов процесса. В общем случае, в ходе сопоставления локальных значений общих переменных осуществляется нейтрализация некоторых доказанных акторов, принадлежащих процессу.

Сорт глобального значения переменной (a sort of global value of variable) — вспомогательная характеристика, приписываемая текущим глобальным значениям общих переменных — «защищённое» или «незащищённое».

Сорт значения порта (a sort of value of port) — вспомогательная характеристика, приписываемая текущим значениям портов — «защищённое» или «незащищённое».

Сорт порта (a sort of port) — вспомогательная характеристика, приписываемая портам; процесс относит каждый из своих портов к одному из трёх сортов: «простой», «отключающий», «защищающий». Сорта портов задаются с помощью описателей портов или по умолчанию.

Сорт потокового сообщения (a sort of flow message) — вспомогательная характеристика, приписываемая потоковым сообщениям. Различаются два сорта потоковых сообщений — «защищённое» и «незащищённое». Потоковое сообщение является защищённым, если оно непустое и было отправлено (передано) через защищающий порт. В остальных случаях потоковое сообщение является незащищённым. В частности, сорт пустого сообщения всегда «незащищённое».

Составной терм (compound term) — структура, список или недоопределённое множество.

Состояние актора (state of actor) — Актор может находиться в одном из трёх состояний: доказанный, активный, нейтральный.

Состояние порта (state of port) — вспомогательное логическое значение, поставленное в соответствие каждому порту. Считается, что порт процесса всегда находится в одном из двух возможных состояний: согласованный или несогласованный.

Состояние процесса (state of process) — Процесс может находиться в одном из трёх состояний: 1) «объявленный; 2) «используемый»; 3) «неиспользуемый». «Используемый процесс» — это обобщающее название для следующих трёх состояний процесса: 1) «сформированный»; 2) «доказанный»; 3) «неудачный».

Спейсер (spacer) — константа #, обозначающая неизвестный элемент данных или мир.

Специальный символ (special symbol) — графический символ, используемый для построения ограничителей.

Список (list) — составной терм, построенный из последовательности (возможно, пустой) аргументов, заключённой в квадратные скобки. В случае если последовательность аргументов списка не является пустой, в его состав может быть включён дополнительный компонент, обозначающий остаток (хвост) списка.

Список задержанных подцелей (list of suspended subgoals) — вспомогательный список подцелей доказательства, исполнение которых было отложено механизмом задержки исполнения подцелей. Считается, что на каждой фазе исполнения процесса используется новый список задержанных подцелей. В начале фазы список задержанных подцелей является пустым.

Стратегия управления (control strategy) — алгоритм управления исполнением программы, определяющий порядок выбора предложений программы и порядок исполнения подцелей в предложениях. Стратегия управления Акторного Пролога («акторный механизм») является расширением стандартной стратегии управления («поиск слева направо в глубину с возвратом»), соответствующей текстуальному упорядочению процедур и вызовов предикатов. Отличиями акторного механизма от стандартной стратегии управления являются возможности повторного доказательства акторов, а также задержки исполнения подцелей.

Строковый литерал (string literal) — разновидность простого терма — последовательность сегментов строки, обозначающая цепочку графических и управляющих символов.

Структура (structure) — составной терм, построенный из функтора и последовательности одного или более аргументов, заключённой в круглые скобки.

Сформированный процесс (formed process) — одно из возможных состояний процесса — характеризуется тем, что пространство поиска и слоты процесса уже созданы, но при этом некоторые акторы процесса ещё ни разу не были доказаны и, следовательно, не согласованы.

Сцепление переменных (chaining of variables) — отождествление (несвязанных) переменных; любое связывание одной из сцеплённых переменных автоматически вызывает такое же связывание всех сцеплённых с ней переменных. В Акторном Прологе область действия операции сцепления переменной всегда ограничена множеством её вхождений, принадлежащих некоторым конкретным акторам.

Текущее значение порта (current value of port) — некоторый вспомогательный терм, поставленный в соответствие порту.

Терм (term) — синтаксическая конструкция, обозначающая элемент данных или экземпляр класса. Различаются простые термы, составные термы, а также вызовы функций в предложениях.

Точка выбора (backtrack point) — неисследованный путь, по которому может пойти исполнение программы в случае отката.

Унификация (unification) — операция сравнения (отождествления) нескольких формул, связывающая переменные в составе формул сопоставленными с ними подформулами. Унификация различных (несвязанных) переменных приводит к сцеплению этих переменных.

Унифицировать, '===' (unify) — предопределённый предикат языка, вызывающий унификацию заданных аргументов.

Управляющий оператор (control statement) — синтаксическая конструкция, выражающая целостное законченное действие, реализуемое во время исполнения программы и способное нарушить её полноту относительно декларативной семантики.

Управляющий символ (control character) — элемент набора символов ASCII — возврат на одну позицию, горизонтальная табуляция, перевод строки, вертикальная табуляция, перевод формата или возврат каретки.

Фаза исполнения процесса (phase of execution of process) — законченный период исполнения процесса, соответствующий обработке процессом некоторого сообщения или изменению состояния процесса. После успешного окончания фазы исполнения процесса осуществляется фиксирование процесса.

Факт (fact) — предложение, в составе которого отсутствуют подцели.

Фиксирование процесса (fixation of process) — действия, осуществляемые после успешного окончания очередной фазы исполнения процесса: 1) устранение всех точек выбора, возникших в течение этой фазы; 2) фиксирование всех общих переменных всех акторов, принадлежащих процессу.

Фиксирование терма (fixation of term) — заменена спейсером # всех несвязанных переменных в составе терма. В соответствии с семантикой Акторного Пролога, фиксирование не распространяется на переменные в составе миров, являющихся компонентами фиксируемого терма.

Фиксированное значение (fixed value) — значение терма, в котором заменены спейсером # все несвязанные переменные, за исключением переменных в составе миров, являющихся компонентами рассматриваемого терма.

Формирование программы (formation of program) — сборка программы из библиотечных модулей, сопровождаемая проверкой её синтаксической правильности.

Формирование процесса (formation of process) — последовательность действий, создающих пространство поиска и слоты процесса: 1) Доказательство простого конструктора, заданного в составе конструктора процесса. 2) Определение портов процесса. 3) Проверка текущих значений портов процесса и перевод процесса в состояние «используемый сформированный» или в состояние «неиспользуемый» (в соответствии с правилами переключения состояний процесса). 4) Автоматическое отправление процессу инициализирующего потокового сообщения.

Формирование экземпляра класса (formation of class instance) — последовательность действий, создающих пространство поиска и слоты экземпляра класса.

Функтор (functor) — имя (символ или метапеременная), которому приписана некоторая арность (число аргументов). Метапеременная, используемая в качестве функтора, называется метафункто́ром.

Функция (function) — разновидность предиката, предназначенная для имитации подпрограмм-функций, возвращающих выходное значение.

Хвост (tail) — остаток списка или недоопределённого множества.

Целевое утверждение (goal statement) — утверждение, с доказательства которого начинается исполнение программы.

Целевой мир резидента (target world of resident) — один из миров, в которых резидент исполняет заданный вызов функции.

Целевой параметр (target parameter) — переменная или атрибут, обозначающие мир, в котором должен быть исполнен дальний вызов предиката.

Целевой процесс резидента (target process of resident) — один из процессов, состояние которых отслеживает резидент; в состав целевого процесса входят некоторые целевые миры резидента.

Число (number) — разновидность простого терма — числовой литерал, перед которым может стоять знак минус.

Числовой литерал (numerical literal) — лексема, обозначающая числовое значение. Числовые литералы бывают целые и вещественные (плавающие).

Экземпляр класса, мир (class instance, world) — конкретное применение класса; составная часть пространства поиска исполняемой программы. Экземпляр класса характеризуется набором предложений соответствующего класса и его предков, а также набором слотов, доступных во всех этих предложениях. Построение экземпляра класса осуществляется в результате доказательства утверждения о его существовании (конструктора).

Элементарный программный модуль (basic program unit) — определение класса или определение проекта.

Элемент данных (data item) — обозначенная группа данных, обрабатываемая как единое целое.

Приложение D

Список понятий языка

(Данное приложение не является частью определения языка.)