Пример 3-22. Подсписок.

Декларативный смысл: **SubList** является подсписком **List**, если выполняются следующие условия:

1. **List** может быть разложен на некоторый список и список **List2**.
2. **List2** может быть разложен на **SubList** и некоторый список.

Предложенные варианты процедуры **sub** позволяют найти все имеющиеся в исходном списке подсписки.

Вариант 1. На рис. 3 показано отношение **sub** для 1-го варианта.

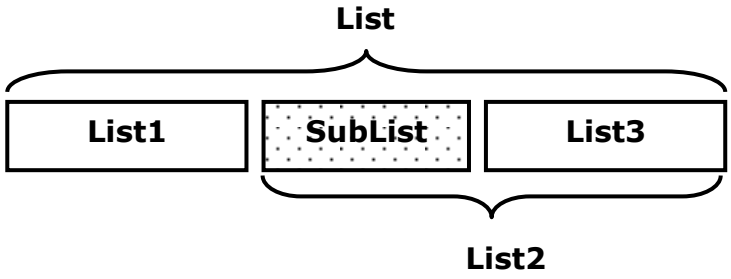

Рис. 3. Отношение sub (1-ый вариант)

```
domains     list = symbol*
predicates  nondeterm sub (list, list)
            nondeterm append (list, list, list)
clauses     sub (List, Sublist) :-
                append (_, List2, List), append (Sublist, _, List2 ).
            append ( [ ], List, List ).
            append ( [X|L1], L2, [X|L3] ) :- append (L1, L2, L3).
goal        sub ([a, b, c, d], X).
```

Вариант 2. На рис. 4 показано отношение **sub** для 2-го варианта.

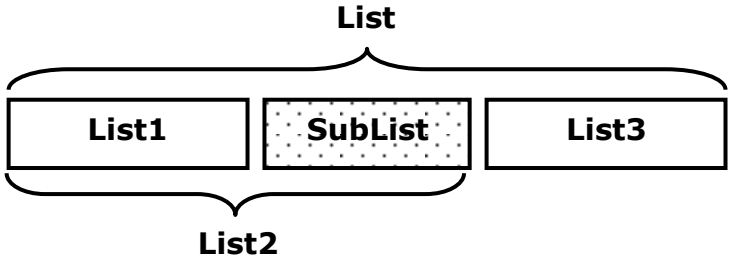

Рис. 4. Отношение sub (2-ой вариант)

```
domains     list = symbol*
predicates  nondeterm sub (list, list)
            nondeterm append (list, list, list)
clauses     sub (List, Sublist) :-
                append (List1, _, List), append (_, Sublist, List1).
            append ( [ ], List, List ).
            append ( [X|L1], L2, [X|L3] ) :- append (L1, L2, L3).
goal        sub ([a, b, c, d], X).
```

Решения для вариантов:

1-ый вариант правила **sub**	2-ой вариант правила **sub**
X=[]	**Y=[]**
X=["a"]	**Y=["a"]**
X=["a","b"]	**Y=[]**
X=["a","b","c"]	**Y=["a","b"]**
X=["a","b","c","d"]	**Y=["b"]**
X=[]	**Y=[]**
X=["b"]	**Y=["a","b","c"]**
X=["b","c"]	**Y=["b","c"]**
X=["b","c","d"]	**Y=["c"]**
X=[]	**Y=[]**
X=["c"]	**Y=["a","b","c","d"]**
X=["c","d"]	**Y=["b","c","d"]**
X=[]	**Y=["c","d"]**
X=["d"]	**Y=["d"]**
X=[]	**Y=[]**
15 Solutions	**15 Solutions**

3.3.4. Компоновка данных в список

Из имеющихся в программе фактов можно формировать списки с целью последующей обработки исходных данных. Для этого существует встроенный предикат **findall**, который имеет следующий формат:

findall (VariableName, PredicateExpression, ListName)

, где **VariableName** – имя переменной, значение которой необходимо собрать в список,

PredicateExpression – предикат, из которого нужно собирать значения,

ListName – имя списка значений, собранных методом поиска с возвратом.

Пример 3-23. Использование предиката **findall**.

```
domains    name = string                    % фамилия
           age  = integer                   % возраст
           list = age*                      % список возрастов
           quantity = integer               % длина списка
predicates nondeterm person (name, age)     % сотрудник
           % вычисление суммы возрастов и длины
           sumlist (list, age, quantity)
           run                              % цель
goal       run.
clauses    % факты
           person ("Иванов", 42).
           person ("Петров", 36 ).
           person ( "Сидоров", 27 ).
```

```
% вычисление суммы возрастов и длины
sumlist ([ ], 0, 0).
sumlist ([ H | T ], Sum, N) :–
        sumlist (T, Si, Ni), Sum = H + Si, N = 1 + Ni.
% цель
run :–
        % формирование списка возрастов
        findall (Age, person ( _, Age ), List),
        % вычисление суммы возрастов и длины списка
        sumlist (List, Sum, LenList),
        % вычисление и вывод среднего возраста
        AverageAge = Sum / LenList,
        write ("Средний возраст = ", AverageAge),
        exit.
```

4. ВСТРОЕННЫЕ ПРЕДИКАТЫ ОБРАБОТКИ СТРОК

Отличительной особенностью языка *Пролог* является эффективность его использования для реализации символьных преобразований. Это обеспечивается наличием типа данных *список* и богатого набора встроенных предикатов символьных преобразований, которые разбиты на группы:

- Разделение строк на части *(подстроки или лексемы)*.
- Синтез строк из подстрок или лексем.
- Проверка вхождения в строку подстрок или лексем.
- Преобразование строк.
- Вычисление или проверка длины строки.
- Создание пустой строки заданной длины.
- Проверка, является ли строка именем.
- Форматный вывод в строку.

Аргумент, который присваивается или назначается в момент вызова, называется входным аргументом и обозначается буквой **i** (input).

Свободный аргумент – это выходной аргумент, который обозначается буквой **o** (output).

4.1. Создание символьных префиксов

Создать символьный префикс – это значит присоединить этот символ к началу строки.

Формат предиката:

frontchar (String, Char, Rest)

, где **String** – строка,
Char – первый символ в строке,
Rest – строка без первого символа.

Предикат **frontchar** действует согласно равенству:

String = Char + Rest

Предикат терпит неудачу, если строка **String** связана со строкой нулевой длины.

Можно использовать данный предикат:

1. Для расщепления строки в последовательность символов.
2. Для создания строки из последовательности символов.
3. Для проверки символов в строке.

В табл. 10 показаны примеры использования предиката **frontchar**.

Таблица 10

Примеры использования предиката frontchar для разных потоков данных

Потоки данных	String	Char	Rest	Результат
(i, o, o)	"доклад"	**C**	**R**	**C** = 'д' **R** = "оклад"
(i, i, o)	"доклад"	'д'	**R**	**R** = "оклад"
(i, o, i)	"оклад"	**C**	"клад"	**C** = 'о'
(o, i, i)	**S**	'к'	"лад"	**S** = "клад"
(i, i, i)	"клад"	'к'	"лад"	**yes**

4.2. Формирование лексем

Последовательность символов является лексемой, если она представляет:

- имя в соответствии с синтаксисом языка *Пролог*;
- число (причём, *знак числа является отдельной лексемой*);
- отличный от пробела знак.

Формат предиката:

fronttoken (String, Token, Rest)

, где **String** – строка,
Token – лексема в начале строки,
Rest – строка без первой лексемы.

Предикат **fronttoken** действует согласно равенству:

String = Token + Rest

Можно использовать данный предикат:

1. Для расщепления строки в последовательность лексем.
2. Для создания строки из последовательности лексем.
3. Для проверки лексем в строке.

В табл. 11 показаны примеры использования предиката **fronttoken** для разных потоков данных.

Таблица 11

Примеры использования fronttoken для разных потоков данных

Потоки данных	String	Token	Rest	Результат
(i, o, o)	"Road"	**T**	**R**	**T** = "Road" **R** = ""
(o, i, i)	**S**	"Rail "	"Road"	**S** = "Rail Road"
(i, i, o)	"Rail Road"	"Rail"	**R**	**R** = " Road"
(i, o, i)	"Rail Road"	**T**	" Road"	**T** = "Rail"
(i, i, i)	"Rail Road"	"Rail"	" Road"	**yes**

4.3. Создание подстрок

Формат предиката:

frontstr (N, String, StartString, EndString)

, где **String** – строка,
StartString – подстрока из **N** первых символов строки **String**,
EndString – остаток строки **String**.

В табл. 12 показан пример использования предиката **frontstr**.

Таблица 12

Пример использования предиката frontstr

Поток данных	N	String	StartString	EndString	Результат
(i, i, o, o)	2	"OS Unix"	**Start**	**End**	**Start** = "OS" **End** = " Unix"

4.4. Конкатенация строк

По крайней мере, два аргумента должны быть связаны при вызове предиката. Таким образом, этот предикат даёт всегда только одно решение.

Формат предиката:

concat (String1, String2, String3)

, где **String3** является сцеплением строк **String1** и **String2**.

В зависимости от потоков данных предикат **concat** действует по формулам, представленным в табл. 13.

Таблица 13

Формулы действия предиката concat для разных потоков данных

Поток данных	Формула предиката
(i, i, o)	String3 = String1 + String2
(i, o, i)	String2 = String3 – String1
(o, i, i)	String1 = String3 – String2

Предикат с потоком данных *(i, i, i)* проверяет, является ли **String3** строкой, полученной сцеплением строк **String1** и **String2**.

4.5. Длина строки

Формат предиката: **str_len (String, Length)**

, где **String** – строка,
Length – длина строки **String**.

Назначение предиката **str_len** зависит от потоков данных так, как показано в табл. 14.

Таблица 14

Зависимость назначения предиката str_len от потока данных

Поток данных	Назначение предиката
(i, o)	Определяет длину строки.
(i, i)	Проверяет длину строки.
(o, i)	Возвращает строку пробелов заданной длины.

4.6. Проверка строки на имя

Имя в *Прологе* начинается с буквы или символа подчёркивания.

Формат предиката: **isname (String)**

, где **String** – проверяемая *на имя* строка.

Поток данных этого предиката *(i)*. Возвращает истину, если проверяемая строка является именем. Иначе – ложь.

4.7. Преобразование данных

Поток данных *(i, i)* во всех предикатах служит для проверки преобразования и возвращает истину, когда преобразование верно (иначе – ложь).

Стандартные предикаты преобразования для остальных потоков данных представлены в табл. 15.

Таблица 15

Стандартные предикаты преобразования типов

Формат предиката	Поток данных	Преобразование
char_int (Char, Integer)	(i,o)	Символ в цифру.
	(o, i)	Цифра в символ.
str_char (String, Char)	(i,o)	Строка в символ. Неудача, если String = "" или длина строки больше 1.
	(o, i)	Символ в строку.
str_int (String, Integer)	(i,o)	Строка в целое число. Неудача, если String = "".
	(o, i)	Целое число в строку.
str_real (String, Real)	(i,o)	Строка в вещественное число. Неудача, если String = "".
	(o, i)	Вещественное число в строку.
upper_lower (Upper, Lower)	(i,o)	Строку или символ из верхнего регистра в нижний регистр.
	(o, i)	Строку или символ из нижнего регистра в верхний регистр.

Пример 3-24. Преобразование строки в список лексем.

Декларативный смысл: Список состоит из элементов, каждый из которых представляет собой лексему входной строки.

```
domains
        tok = number (integer) ;
              name (string) ;
              char (char)
        list = tok*
predicates
        scanner (string, list)     % преобразования строки в список лексем
        makeTok (string, tok)      % преобразование лексемы в элемент списка
clauses
        % преобразования строки в список лексем
        scanner ( "", [ ] ) :- !.
        scanner ( String, [Tok | Tail] ) :–
              fronttoken (String, Lexeme, Rest),
              makeTok (Lexeme, Tok),
              scanner (Rest, Tail).
        % преобразование лексемы в элемент списка
        makeTok (L, name (L)) :– isname (L), !.
        makeTok (L, number (N)) :– str_int (L, N), !.
        makeTok (L, char (C)) :– str_char (L, C), !.
```

```
goal
        write ("Введите текст:\n"), readln (Text),
        scanner (Text, TokList),
        write (TokList), exit.
```

Если введён текст:

Text = "I know 5 names of the girls."

Решением будет:

TokList = [name ("I"), name ("know"), number (5), name ("of"), name ("the"), name ("girls"), char ('.')]

Для формирования списка, который состоит из элементов разного типа, в программе используется множественное объявление объектов с доменным именем **tok**.

Инициализация выходного списка нулевым списком происходит по первому правилу процедуры **scanner**, когда входная строка пустая.

Второе правило процедуры **scanner** последовательно выделяет очередную лексему из строки, преобразует эту лексему в голову **Tok** списка при помощи предиката **makeTok**, и рекурсивно вызывает предикат **scanner**, передавая в качестве аргументов остаток строки **Rest** и хвост списка **Tail**.

На выходе из рекурсии элементы **Tok**, размещённые в стеке, будут присоединены к списку как головы согласно второму правилу процедуры **scanner**.

5. УПРАВЛЕНИЕ ХОДОМ ВЫПОЛНЕНИЯ ПРОГРАММЫ

Программирование на *Прологе* заключается в определении отношений, и в постановке касающихся этих отношений целей. *Пролог*-программа состоит из предложений, которые могут быть фактами, правилами и целями. Ответ на поставленную цель представляет собой **множество объектов**, которые удовлетворяют цели.

Отношение может определяться:

- фактами, перечисляющими объекты, для которых это отношение *выполняется*;
- правилами, содержащими утверждения, истинность которых *зависит* от некоторых условий.

Процесс, в результате которого *Пролог*-система устанавливает, удовлетворяет ли объект цели, включает:

- логический вывод,
- исследование различных вариантов,
- при необходимости откат *(backtracking)*.

Для решения поставленной задачи автор программы пишет процедуры (процедура – множество правил об одном и том же отношении).

Порядок следования правил в процедуре, как и порядок следования подцелей внутри правил, часто влияет на скорость поиска решений. Следовательно, важен способ получения решения.

5.1. Декларативный и процедурный смысл программы

Декларативный смысл определяет, **что** должно быть результатом работы программы, и касается **только** представленных в программе **отношений.**

При составлении программы на *Прологе* следует установить, какие факты и отношения свойственны решаемой задаче. Они могут быть непосредственно выражены утверждениями и образуют декларативное описание задачи. Такое описание не задаёт порядок использования утверждений и не показывает, как они могут быть использованы для решения задачи. *Пролог* будет пытаться согласовать утверждения сверху вниз и слева направо.

Декларативный смысл программы позволяет установить, является ли заданная цель истинной, и если да, то при каких значениях переменных она достигается.

Процедурный смысл определяет, **как** результат был получен, то есть, как отношения реально обрабатывались *Пролог*-системой.

Возможно, для достижения результата придётся добавить некоторые детали для управления ходом выполнения программы, которые укажут системе то, как использовать факты и отношения. Полученная в результате программа рассматривается как процедурная.

При написании процедуры необходимо учитывать следующее:

1. Сколько раз может быть успешно согласовано целевое утверждение? Если механизм возврата снова приведёт к этому целевому утверждению, возможно ли его новое согласование и нужно ли оно?
2. Какая конкретизация переменных происходит при использовании каждого утверждения? Смысл вызова процедуры заключается в том, чтобы произвести какую-либо проверку и/или конкретизацию или сцепление переменных.
3. Какие параметры используются для входных значений, и какие – для выходных? Некоторые процедуры могут успешно работать с разными потоками данных.

Получить ответ на вопрос относительно процедуры означает попытаться достичь целей, заданных в списке. Их можно достичь, если переменные, которые встречаются в целях, могут быть конкретизированы таким образом, что цели логически следуют из программы. В случае неудачи конкретизация отсутствует.

Формально два подхода можно представить предложением вида:

P :– Q, R.

Декларативный смысл: **P** истинно, если подцели **Q** и **R** истинны.

Процедурный смысл: Чтобы решить задачу **P**, нужно сначала решить подцель **Q**, а затем – подцель **R**.

Основное различие между декларативным и процедурным смыслом заключается в том, процедурный смысл определяет *не только логические связи* между предложениями, *но ещё и порядок*, в котором эти предложения должны обрабатываться.

Пример 5-1. Определение декларативного и процедурного смысла для процедуры проверки принадлежности элемента списку.

Декларативный смысл: Элемент принадлежит списку, если он совпадает с головой списка, или если он принадлежит хвосту списка.

Процедурный смысл: Чтобы узнать, принадлежит ли элемент списку, нужно сравнить его с головой списка. Если элемент – не голова списка, то проверить хвост списка на наличие в нём искомого элемента.

5.2. Процесс решения задачи в Прологе

Поиск решения начинается с процесса согласования цели, и состоит из следующих шагов:

1. *Пролог*-система ищет предложение программы с таким же именем предиката, как и у целевого утверждения.
2. *Пролог*-система проверяет цель и найденное предложение на соответствие количества и типов аргументов.
3. Если первые два условия выполнены, происходит попытка согласования аргументов по правилам согласования объектов (раздел 2.9). Результатом удачного согласования будет конкретизация объектов целевого утверждения.

   Конкретизация – это процесс получения переменной значения в результате подстановки вместо переменной конкретного объекта.

4. Если предложение согласования является правилом, *Пролог*-система переходит к согласованию тела правила (раздел 2.11).

   Если тело правила пустое, то целевое утверждение будет решено.

   Если же тело правила не пустое, то система помещает в стек каждую входящую в тело подцель и обрабатывает её точно так же, как исходную цель. Подцели проверяются слева направо и сверху вниз. Если все подцели из тела правила будут успешно обработаны, то целевое утверждение тоже будет успешным.

5.3. Принудительный возврат

В процессе достижения цели осуществляется автоматический перебор вариантов, для чего используется механизм обратного просмотра:

```
domains    name = symbol
facts      father (name, name)
clauses    father (carl, clara).    father (carl, tom).
goal       father (X, Y), write (X, " is "), write (Y, "'s father").
```

Решение:

```
carl is clara's father
X=carl, Y=clara
carl is tom's father
X=carl, Y=tom
2 Solutions
```

Если добавить в конец цели предикат **exit**, чтобы избежать вывода значений по умолчанию (**X=carl, Y=clara** и т.д.), то перебора не будет, так как **exit** запрещает перебор, и ищется только одно *(самое первое)* решение.

```
goal       father (X, Y), write (X, " is "), write (Y, "'s father"), exit.
```

Решение:

```
carl is clara's father
```

Возможность управления механизмом поиска с возвратом предоставляет *предикат принудительного возврата*.

Формат предиката принудительного возврата:

fail

Предикат **fail** никогда не может быть согласован, поэтому *Пролог* вынужден выполнять поиск с возвратом. Помещать подцель после **fail** в правиле бесполезно, так как предикат всё время завершается неудачно, а значит, нет возможности для достижения расположенной после **fail** подцели.

Предикат **fail**:

- никогда не согласуется,
- вынуждает *Пролог*-систему выполнять поиск с возвратом,
- расположенные после предиката подцели не могут быть достигнуты.

После замены в цели **exit** на предикат **fail** результат изменится.

```
goal       father (X, Y), write (X, " is "), write (Y, "'s father\n"), fail.
```

Решение:

```
carl is clara's father
carl is tom's father
No Solution
```

Чтобы программа заканчивалась успешно, необходима процедура:

```
predicates  run
clauses     run :-
                father (X, Y), write (X, " is "), write (Y, "'s father\n"),
                fail.
            run.
goal        run, exit.
```

После вывода всех найденных решений по первому правилу **run** и последней неудачи, управление получит альтернативное второе правило **run**, которое всегда согласуется. Программа завершится успехом.

Решение:

```
carl is clara's father
carl is tom's father
```