SPARD

String processing Pattern matching Rule-based Data-driven language

Спецификация языка

Версия 0.51

Автор – Хиль В.А.

По всем вопросам и предложениями просьба обращаться по следующим контактам:

Веб-сайт: http://vladimirkhil.com

Email: vladimir.khil at gmail dot com

Содержание

Введение. Основные понятия и определения

Язык SPARD

Эта спецификация описывает язык SPARD, его общие принципы, механизм работы SPARD-преобразователя и полный синтаксис языка. Любой преобразователь, выполняющий преобразования в точности по всем правилам, изложенным в этом документе, считается корректной реализацией языка SPARD.

SPARD – язык обработки потоковых данных (списков, текстов, иерархических объектов и др.). Обработка данных в языке SPARD описывается при помощи набора правил, задающих отображение этих данных в произвольный результат.

Цель языка – предоставление пользователю средств для описания сложных преобразований простым способом без необходимости серьёзной подготовки к этому.

Потоковые данные (поток) – последовательности (конечные или бесконечные) однородных объектов произвольного типа. Понятие однородности определяется неформально. Объекты называются **однородными**, если к ним могут быть применены только общие операции, т.е. если некоторая операция применима к одному из таких объектов, то она же обязательно применима и к другому. Однородные объекты составляют **множество однородных данных** (**тип данных**).

Примерами однородных объектов могут служить символы и строки (с операцией конкатенации), числа (с арифметическими операциями) и графы (например, с операцией построения остовного дерева). В программировании однородными объектами являются массивы и списки, а неоднородными – структуры.

Экземпляры объектов, составляющие поток, называются **элементами** этого потока.

Примерами потоковых данных могут служить:

1) текст на естественном языке («Каждый охотник желает знать, где сидит фазан») – последовательность символов;

2) XML-файл (<elem attr=”value”><internal /></elem>) – последовательность узлов документа;

3) текст на языке программирования (for (int i = 0; i < 5; i++) { Console.WriteLine(i); });

4) последовательность целых чисел (`4, 8, 15, 16, 23, 42`).

Выполнение программы на языке SPARD рассматривается как преобразование одного потока данных в другой, причём тип объектов результирующего потока не обязан совпадать с типом объектов исходного. Данная модель не ограничивает общность описания преобразований, так как произвольный объект можно считать элементом потока длиной 1.

Преобразуемый поток называется **входным потоком данных**, результирующий – **выходным потоком данных**.

Произвольная конечная подпоследовательность входного (выходного) потока называется **цепочкой входных (выходных) данных** или **входной (выходной) цепочкой**.

Текущая версия языка SPARD адаптирована в первую очередь под **символьные преобразования** (преобразования текста в текст). Элементами входного и выходного потоков (текста) в таком сценарии являются **символы**, а множеством данных – **алфавит** языка. При описании текстовых преобразований в данном документе понятия «элемент» и «символ», а также «множество входных данных» и «алфавит» эквивалентны.

Типичные задачи преобразования текстов, решение которых может быть упрощено посредством использования SPARD[1], перечислены ниже:

- поиск (рассматриваемый как преобразование текста запроса в результаты поиска);
- извлечение информации;
- автореферирование;
- создание вопросов по тексту;
- форматирование текста (в частности, исправление ошибок);
- компиляция;
- преобразование программ и структурированных данных (метапрограммирование);
- символьные вычисления (компьютерная алгебра);
- перевод (компьютерная лингвистика);

[1] по сравнению с использованием широко распространённых языков программирования

- генерация отчётов.

Отличительные особенности синтаксиса языка SPARD:

1. Оптимизация под обработку текстов.
2. Краткость.
3. Простота для неспециалиста в области компьютерных технологий (например, лингвиста).

SPARD-программа

В самом общем виде **формальная модель программы** на языке SPARD представляет собой упорядоченный набор **правил** преобразования $R_1, R_2, \dots, R_n$. Каждое правило $R_k (k = 1, \dots, n)$ состоит из **предиката** I_k, определённого на произвольных цепочках входных элементов, и **функции** (**порождения**) **результата** O_k, возвращающую цепочку выходных элементов. Ниже будет дано строгое определение различных частей программы.

Предикат и результат могут быть параметризованы при помощи параметрического вектора $p_k = (p_{k1}, p_{k2}, \dots, p_{kN_k})$, где N_k – число параметров для правила R_k (длина вектора). Каждый из параметров – это переменная, способная принимать значения из некоторого заданного множества (соответственно $P_{k1}, P_{k2}, \dots, P_{kN_k}$). Обозначим $P_k = P_{k_1} \times P_{k_2} \times \dots \times P_{k_{N_k}}$ – множество значений параметрического вектора.

Таким образом, предикат I_k для цепочки, состоящей из элементов множества A, определён как:

$$I_k : A *\times P_k \to \{T, F\}$$

Важной функцией является функция $Test$ проверки цепочки предикатом. Для каждой цепочки выбирается некоторый её непустой префикс, на котором предикат истинен при определённых значениях параметров. Функция $Test$ возвращает упорядоченное множество всех возможных вариантов выполнения предиката на цепочке. Каждый вариант – это пара из постфикса входной цепочки (та часть, что вместе с префиксом составляет исходную цепочку) и значения параметрического вектора, при котором предикат истинен на некотором префиксе. Если не существует ни одного варианта для истинности предиката, функция возвращает пустое множество.

$$Test(I_k, \alpha) = \{(\delta, p) |\, \alpha = \gamma\delta, |\gamma| > 0, I_k(\gamma, p) = T\}$$

Порядок пар определяется типом предиката и будет рассмотрен позднее.

Функция результата O_k возвращает результирующую цепочку, состоящую из элементов множества B. Она определяется как:

$$O_k : P_k \to B*$$

Тем самым, функция результата напрямую не зависит от входной цепочки. Всё влияние входа на результат вычисления должно быть передано через значения параметров.

Полностью формализованное определение алгоритмической модели SPARD выходит за рамки данного документа.

SPARD-преобразователь

Выполнение программы, написанной на SPARD, и собственно само преобразование определяются через механизм работы абстрактного вычислительного устройства, называемого **SPARD-преобразователем**.

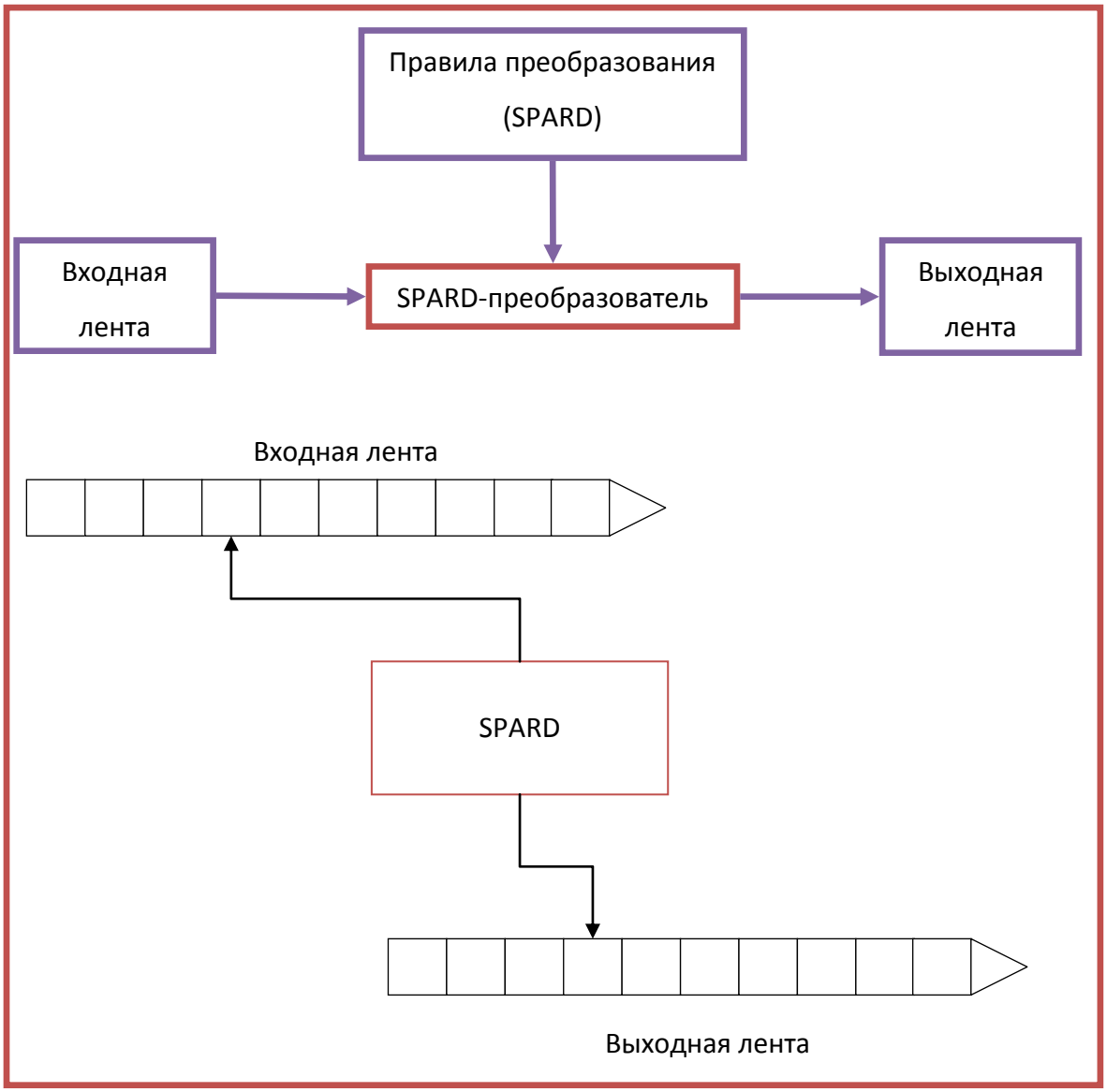


Рисунок 1. SPARD-преобразователь (два представления)

Преобразователь содержит:

- **входную ленту**, на которой находится входной поток данных. Лента состоит из ячеек, в каждой из которых занесён символ (элемент) некоторого алфавита (множества). Лента также имеет считывающую головку, способную находиться напротив одной из ячеек, считывать содержащийся в ней элемент и перемещаться к следующей ячейке на ленте. Входная лента потенциально бесконечна;
- **выходную ленту**, на которую записывается результат преобразования – поток выходных данных. Она также состоит из ячеек и записывающей головки, способной находиться напротив одной из ячеек, записывать в неё один результирующий элемент и

перемещаться к следующей ячейке на ленте. Выходная лента также потенциально бесконечна;

- правила преобразования, написанные на SPARD (**программа**) и управляющие работой преобразователя.

Позиция входного потока данных – положение головки, считывающей этот поток. Текущей позиции соответствует **текущий элемент**, обозреваемый в данный момент преобразователем (элемент, находящий в ячейке, напротив которой расположена головка).

Конкретный преобразователь, выполняющий преобразования по правилам, описанным в данном документе, называется **реализацией SPARD**.

Принцип работы преобразователя

Работа преобразователя – последовательность действий, генерирующих выходные данные по входным согласно правилам на языке SPARD. Выполнение действий производится за счёт постепенного считывания данных на входной ленте и записи результата на выходную по мере его формирования.

Перед началом работы считывающая головка на входной ленте установлена напротив первого элемента входных данных, а записывающая головка выходной ленты – в начале выходной ленты. В конце работы преобразователь должен прочитать всю входную ленту и выдать на выходную ленту весь требуемый набор данных в необходимом порядке. Состав выходных данных определяется правилами работы преобразователя.

Работа преобразователя заключается в последовательном выполнении двух элементарных шагов до тех пор, пока все входные данные не будут преобразованы (или не произойдёт ошибки):

1) сопоставление с образцом;
2) вывод частичного результата.

Сопоставление с образцом – это операция проверки входного потока данных в текущей позиции с помощью предиката правила. Суть проверки заключается в

применении предиката ко всем возможным цепочкам входа, начинающимся с текущего элемента.

Общий алгоритм работы преобразователя таков:

1) проверить входной поток данных в текущей позиции предикатами всех правил SPARD, перебирая правила строго в порядке их описания и перебирая для каждого из них все возможные подцепочки, начинающиеся с текущего элемента (в порядке, определяемом типом предиката правила);

2) при истинности предиката правила на некоторой цепочке вывести соответствующий этому правилу результат в выходной поток и продвинуться во входном потоке на длину цепочки, на которой выполнился предикат. Если во входном потоке ещё остались элементы, перейти к шагу 1; иначе завершить работу;

3) при отсутствии совпадений завершить работу с ошибкой.

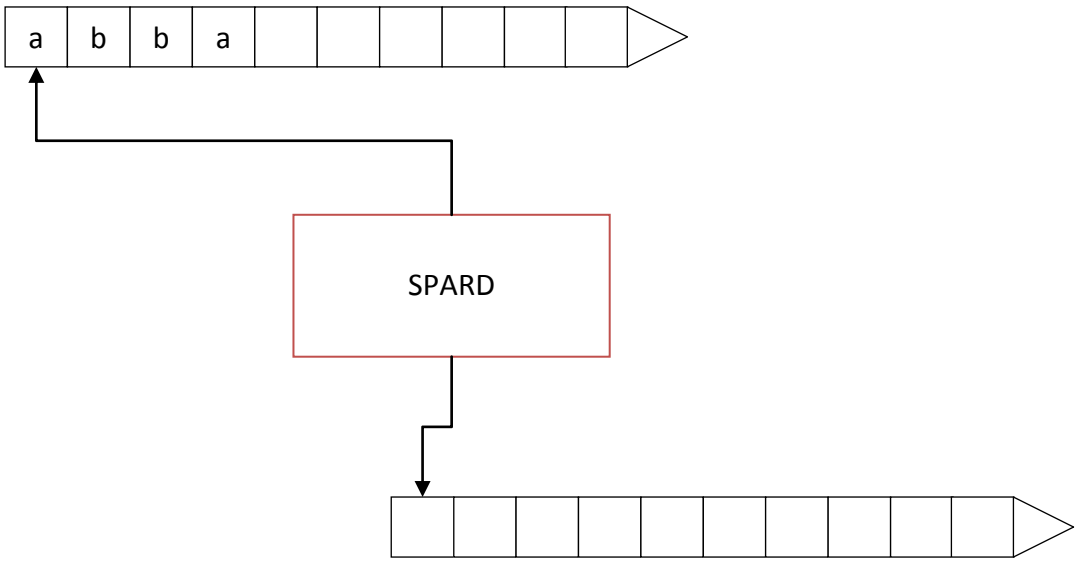
Таким образом, принцип работы преобразователя достаточно прост и полностью определяется правилами SPARD.

Преобразование считается успешным, если преобразователь сумел прочесть все данные на входной ленте и не завершил работу с ошибкой. В противном случае преобразование считается осуществлённым частично.

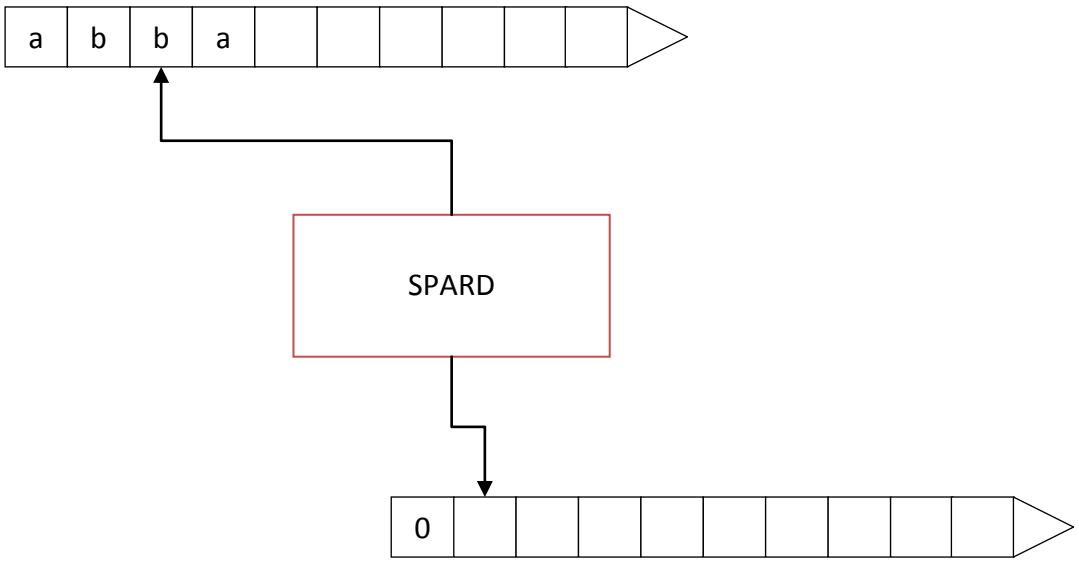
Вот пример работы преобразователя с входом «abba» и набором правил. В данном примере каждое правило располагается на отдельной строке, а знаки => отделяют предикаты от результатов. Предикат истинен только тогда, когда он выполняется на цепочке, которой он обозначается.

```
ab => 0
b => 1
ba => 2
a => 3
```

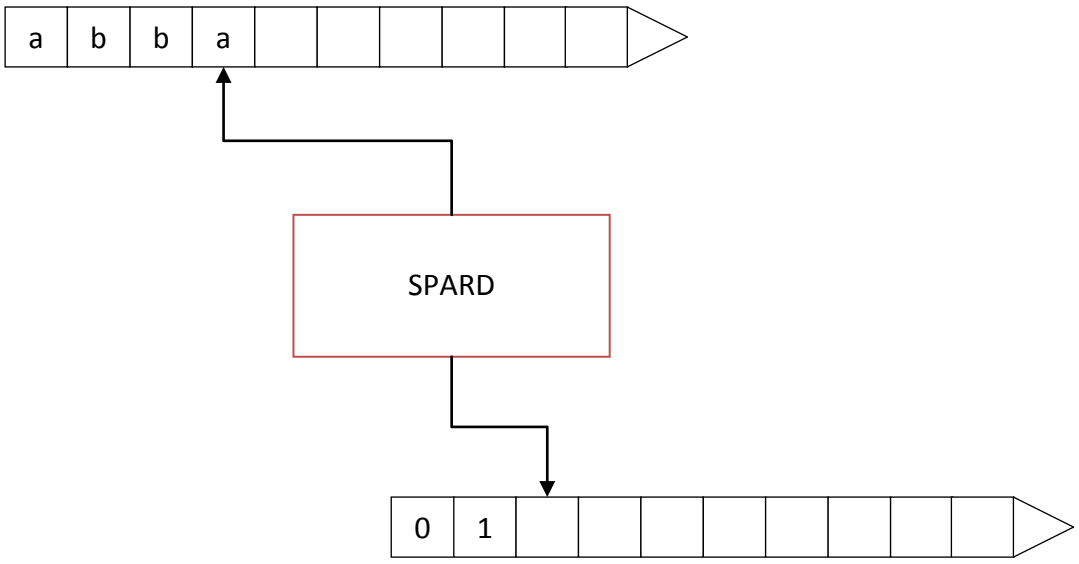
Преобразование будет состоять из следующих шагов.



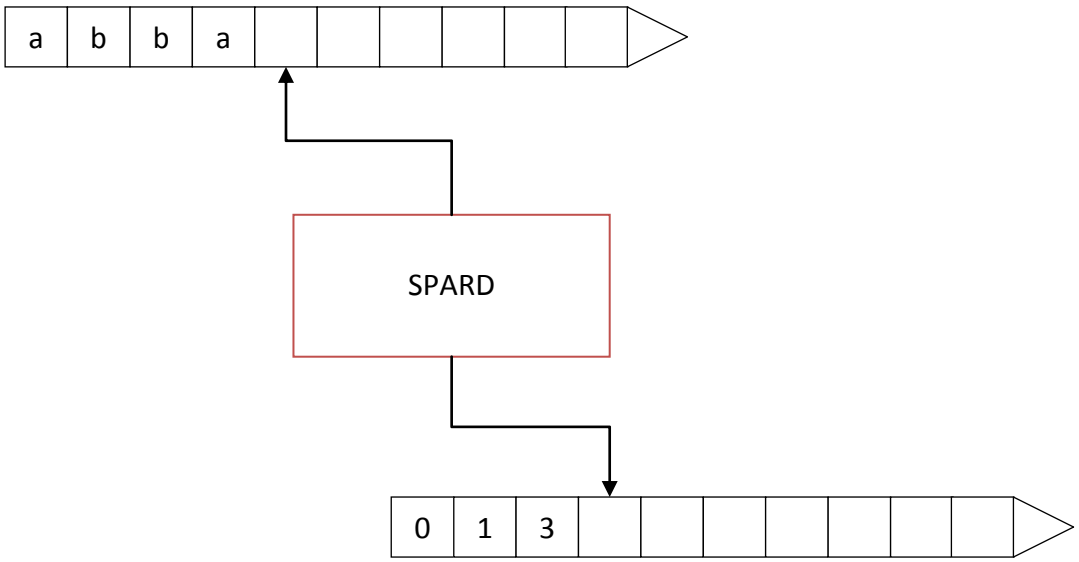
1) В начале работы преобразователь находится в начале входного потока данных и «видит» первым символ “a”. Последовательно перебирая правила, преобразователь начинает сличать входной поток данных с шаблонами (левыми частями правил). В данном примере проверка шаблонов заключаются в проверке строгого совпадения входа в текущей позиции с образцом. Таким образом, будет истинен предикат в правиле (1) (вход начинается с цепочки “ab”), и на выход будет выдан результат (правая часть правила) – символ “0”. С образцом совпало 2 символа, поэтому входной поток данных смещается на две позиции и преобразователь будет «видеть» второй символ “b”;



2) Перебор правил начинается сначала, и первым подходящим правилом будет правило (2) (вход в текущей позиции начинается с “b”). На выход будет выдан символ “1”. Вход сместится на один символ;



3) Последним подошедшим правилом будет правило номер (4). При этом на выход будет выдан символ «3» и вход сместится на одну позицию;



4) Вход полностью исчерпан, поэтому преобразователь завершает свою работу. Результатом преобразования является цепочка «`013`».

Формальное определение SPARD-преобразователя

Пусть α – входная цепочка символов (элементов), занесённая на ленту, и $R = \{R_1, R_2, \dots, R_n\}$ – набор правил (программа). Каждое правило R_k представляет собой пару $< I_k, O_k >$, где I_k– предикат, O_k– функция результата.

Определим функцию I^1 выбора первого элемента упорядоченного множества:

$$I^1(a_1, a_2, \dots) = a_1$$

Для пустого множества I^1 не определена.

Тогда преобразование f и результат преобразования β можно рекурсивно определить так:

$$f(\omega)$$

$$= \begin{cases} O_k(p)f(\delta), \text{если } |\omega| > 0, \exists k \in [1;n]: \begin{cases} Test(I_k, \omega) \neq \emptyset, (\delta, p) = I^1(Test(I_k, \omega)); \\ \forall j \in [1; k-1], Test(I_j, \omega) = \emptyset \end{cases} \\ fail, \text{если } |\omega| > 0 \\ e, \text{иначе} \end{cases}$$

$$\beta = f(\alpha)$$

Первая строка формулы отражает процесс построения частичного результата. Он складывается из последовательного вывода результатов правил O_k (возможно, зависящего от набора параметров p), у которых выполнился предикат ($Test(I_k, \omega) \neq \emptyset$). При этом это первое правило с истинным предикатом в списке ($\forall j \in [1; k-1], Test(I_j, \omega) = \emptyset$).

Вторая строка формулирует условие неудачного преобразования. Полученный частичный результат (на предыдущих шагах преобразования) может быть использован или отброшен в зависимости от специфики задачи.

Последняя строка обозначает условие успешного завершения преобразования (весь вход прочитан) – условие выхода из рекурсии.

Синтаксис языка

Поскольку выразительные возможности языка покрывают возможности форм Бэкуса – Наура и контекстно-зависимых грамматик, а также в целях демонстрации зрелости языка описание синтаксиса SPARD приводится на самом SPARD. Синтаксис во многом похож на РБНФ (конкатенация, оператор выбора `|`) и регулярные выражения (квантификаторы `*`, `+` и `?`), так что он будет интуитивно понятен неподготовленному читателю.

Шаблоны

```
    <Template> := <Primitive> | <Single> | <Binary> | <Polynomial> |
'( <Template> ')
    <Primitive> := <Empty> | <OpenLine> | <CloseLine> | <Any> |
<Anything> | <BasicString>
    <Single> := <Unary> | <Dual>
    <Unary> := <FunctionCall> | <Not> | <ZeroOrOne> | <ZeroOrMore> |
<OneOrMore>
    <Dual> := <Instruction> | <Object> | <Set>
    <Binary> := <Counter> | <Interval>
    <Polynomial> := <Sequence> | <And> | <Or> | <List>
```

Основу выразительности языка составляют шаблоны. Шаблон SPARD – это **выражение**, определяющее множество цепочек данных. Таким образом, каждому шаблону соответствует некоторое (возможно пустое) **множество**. Каждый элемент этого множества представляет собой цепочку (для текстовых преобразований каждый элемент множества – цепочка символов, строка).

Шаблон – некоторая запись, из которой можно получить допустимые цепочки посредством корректных подстановок. К примеру, из шаблона `a.c` можно получить строку `abc` посредством корректной подстановки `b` вместо `.`. Шаблон записывается в виде **выражения** (по аналогии с алгебраическими и регулярными выражениями), содержащего **базовые шаблоны** и **конструкторы шаблонов** (**операции**).

Каждому шаблону соответствует **предикат** (или **предикат шаблона**) – булева функция, истинная на тех и на только тех цепочках данных, которые удовлетворяют шаблону. Предикат является характеристической функцией для множества цепочек, допускаемых шаблоном.

Шаблон может быть параметризован набором параметров $p_1, p_2, \dots, p_n$. В таком случае ему соответствует параметризованные множество и предикат.

Цепочка **совпадает** с шаблоном, если предикат шаблона принимает на этой цепочке значение T (ИСТИНА).

Таким образом, каждому шаблону однозначно соответствует предикат и множество допускаемых цепочек (обратное неверно). В данном документе используется то или иное представление шаблона в зависимости от контекста и удобства. При описании языка для каждого шаблона будут определяться соответствующие ему множество и предикат (для предикатов применяется запись с помощью синтаксиса языка Пролог).

Множественность сравнений и порядок перебора вариантов

При выполнении преобразования входной цепочки SPARD-преобразователь перебирает все возможные префиксы этой цепочки и проверяет совпадение с шаблоном каждого их них. Некоторые шаблоны допускают совпадения сразу с несколькими префиксами в конкретной позиции входных данных. Такие шаблоны называются **множественными**.

Для каждого множественного шаблона задан порядок, в котором преобразователь должен перебирать возможные варианты совпадения. Этот порядок важен, так как различные варианты совпадения могут генерировать отличные друг от друга результаты преобразования.

При сопоставлении входа с шаблоном преобразователь сначала выбирает первый вариант в соответствии с заданным порядком. Если множественный шаблон является частью другого (внешнего) шаблона, преобразователь пытается (используя выбранный вариант) проверить на совпадение оставшуюся часть внешнего шаблона. В случае неудачи преобразователь возвращается к вложенному шаблону и выбирает другой вариант совпадения. Так продолжается до тех пор, пока не произойдёт полное совпадение внешнего шаблона или не будут перебраны все варианты. В последнем случае правило не срабатывает.

Стандартный механизм перебора вариантов для шаблонов можно изменить с помощью *модификаторов*.

Структура программы

```
<Program> := (<SPARDLine><Comment>?<BR>)*
<SPARDLine> := <Definition> | <Rule> | <GlobalInstruction> |
```

Исходный код программы на SPARD состоит из последовательности инструкций, каждая из которых размещается на отдельной строке. Программа может содержать также пустые строки, необходимые для визуального разделения участков кода. Любая строка в программе может заканчиваться необязательным комментарием.

Непустые инструкции SPARD подразделяются на определения, правила и глобальные инструкции. Правила описывают непосредственно саму функцию преобразования исходных данных в результат; определения используются для описания объектов (функций и именованных множеств), задействованных внутри правил; глобальные инструкции служат для настройки преобразователя.

Правила и определения могут включать в себя переносы строк (т.е. располагаться более чем на одной строке). Условия для этого описаны в соответствующем разделе.

Правила

```
<Rule> := <Template> <Direction> <Template>
<Direction> := "=>" | '= | "<="
```

Каждое правило SPARD состоит из шаблона входа, шаблона результата и направления применения правила. В большинстве случаев правило применяется слева направо, т.е. левый шаблон описывает допустимый вход для преобразующей функции, а правый – получаемый в результате преобразования соответствующий данному вход результат.

При применении правила в обратную сторону – налево (<=) – роли шаблонов меняются на противоположные (шаблон входа оказывается расположенным справа, шаблон результата – слева). Направление правила показывает, в какую сторону допустимо его применять. Если направление правила несовместимо с текущим применяемым направлением преобразования (например, правило имеет направление <= при выполнении преобразования слева направо), то правило не может использоваться в данном сценарии.

Знак = означает, что правило может применяться в обоих направлениях. Правило в этом случае называется двусторонним. Двусторонние правила позволяют выполнять обратимые преобразования.

Шаблону входа соответствуют предикат входа и множество цепочек, на которых этот предикат истинен. Предикат проверяется на всех возможных подцепочках входа, начинающихся с текущего элемента. Предикат не может налагать ограничения на прочитанные ранее элементы (исключение составляет только предикат шаблона ^).

Важное требование к шаблону входа – он не может совпадать с пустой цепочкой. Если это было бы возможно, после применения правила текущий элемент остался бы прежним (во входной ленте произошёл бы сдвиг на 0 элементов). В этом случае текущее правило применялось бы снова и снова и преобразователь бы зациклился.

Шаблону результата всегда соответствует множество из одной цепочки. Эта цепочка и возвращается в качестве результата преобразования. Требование единственности цепочки в результирующем множестве накладывает ограничения на синтаксис шаблона результата.

Для двустороннего правила обе его части должны соответствовать правилам синтаксиса результата (так как они строже).

При выполнении преобразования правила перебираются в порядке их определения в исходном тексте программы. Поэтому порядок следования правил влияет на получаемый результат.

Определения

```
<Definition> := <SetDefinition> | <FunctionDefinition>
```

Определения подразделяются на определения именованных множеств и определения функций. Их синтаксис несколько отличается друг от друга. Подробнее определения описаны в разделах про именованные множества и функции.

Определения могут перемежаться с правилами SPARD в тексте программы. Их порядок (в отличие от правил) никак не влияет на получаемый результат преобразования (единственное исключение – порядок следования нескольких определений одного и того же именованного множества).

Комментарии

```
<Comment> := '; <String>[line]$
```

Комментарии начинаются с символа `;` и продолжаются до конца строки. Преобразователь SPARD полностью их игнорирует, они предназначены исключительно для человека.

Простейшие символы и их роль в языке SPARD

Набор символов

В программе на SPARD допускается использовать любые символы Юникода.

Перенос строки

Символы переноса строки отделяют инструкции SPARD друг от друга. Каждая инструкция SPARD располагается на отдельной строке (сами инструкции при этом могут занимать несколько строк).

Пробельные символы

Пробельные символы, находящиеся вне кавычек `""` и не после одинарной кавычки `'`, полностью игнорируются SPARD-преобразователем. Они используются лишь для улучшения читаемости правил преобразования. Пробельные символы можно добавлять в текст программы где угодно. Запрещается лишь помещать пробельные символы внутри двухсимвольных операторов языка (к примеру, `:=`).

Круглые скобки

Круглые скобки используются в SPARD точно так же, как и в других языках. Они позволяют менять порядок сопоставления шаблонов. К примеру, шаблон `a|bc` совпадает со строками `a` и `bc`, а шаблон `(a|b)c` – со строками `ac` и `bc`.

Базовые шаблоны

Этот раздел содержит описания шаблонов, не состоящих из других шаблонов.

Пустой шаблон

```
<Empty> :=
```

Определение

$$()(x) : -True.$$

Пустой шаблон совпадает с пустой цепочкой. Как уже было сказано, шаблон входа не может совпадать с пустой подцепочкой, поэтому шаблон входа не может быть пустым. Пустой шаблон может использоваться как вложенный шаблон и как шаблон входа в

описаниях функций (где ограничения на «непустоту» совпавшей цепочки нет). Пустой шаблон обозначается с помощью отсутствия каких-либо символов (пробельные символы не учитываются):

```
=> result
```

Пустой шаблон допустим в качестве шаблона результата, где он соответствует пустой цепочке (т.е. отсутствию значимого результата):

```
template =>
```

Несмотря на отсутствие результата, правило при этом считается применённым успешно.

Символ (текстовый элемент)

Определение

$$a(x) :- x = 'a'.$$

Предикат шаблона в виде одиночного символа истинен в том и только в том случае, когда текущим символом входного текста является этот символ.

Поскольку многие символы в строке шаблона выполняют служебные функции, перед нецифровыми и небуквенными символами следует ставить апостроф – одинарную кавычку (`'`). Апостроф отменяет любое специальное значение символа в правиле, требуя трактовать его как обычный символ. В частности, с помощью апострофа можно потребовать наличия во входном тексте символа пробела, который в обычной записи игнорируется преобразователем.

Примеры использования шаблона

Шаблон	Истина	Ложь
a	a	b
b	bcd	efg
'.	.	;
'		\t

Любой символ (.)

```
<Any> := '.
```

Определение

$$.(x) :- length(x, 1).$$

Предикат шаблона . истинен на любом одиночном входном текстовом символе. При помощи модификатора line можно изменить логику работы этого предиката: он не будет совпадать с символами возврата каретки (\r в C++) и переноса строки (\n в C++).

Примеры использования шаблона

Шаблон	Истина	Ложь
.	a	\r
.	bcd	\ncd

Начало строки (^)

```
<OpenLine> := '^
```

Предикат шаблона, обозначаемого символом ^, выполняется только в том случае, если головка входной ленты находится в начальной позиции (ни один элемент входа ещё не был прочитан). Таким образом, для символьных преобразований этот предикат истинен лишь в начале текста. При использовании модификатора line предикат также становится истинным в начале любой строки (если последним прочитанным символом был \r или \n).

Если перенос строки обозначается парой символов \r\n, предикат принимает значение ИСТИНА только после символа \n.

При выполнении предиката данный шаблон считается совпавшим с пустой цепочкой.

Примеры использования шаблона (· обозначает текущую позицию во входном потоке)

Шаблон	Истина	Ложь
[line]^	\n·a	a·b
[line]^	\r·t	a·\n
[line]^	\r\n·bcd	\r·\n
[line]^	\n·\n	x·\ny

Конец строки ($)

Предикат шаблона $ истинен только тогда, когда весь входной поток полностью прочитан или (при использовании модификатора line) когда текущим символом входного потока являются символы \r или \n. Если перенос строки обозначается парой \r\n, предикат принимает значение ИСТИНА только перед символом \r.

При выполнении предиката данный шаблон считается совпавшим с пустой цепочкой.

Примеры использования шаблона (· обозначает текущую позицию во входном потоке)

Шаблон	Истина	Ложь
[line]$	a·\n	a·b
[line]$	t·\r	\n·a
[line]$	bcd·\r\n	\r·\n
[line]$	\n·\n	x\n·y

Произвольный вход (_)

```
<Anything> := '_
```

Определение

$$\_(x) : -True.$$

Предикат шаблона _ позволяет принять любые входные данные без учёта того, что именно подаётся на вход. Предикат всегда принимает значение ИСТИНА и в качестве совпавшей цепочки принимается весь непрочитанный остаток входного потока (в этом состоит отличие предиката от пустого). Предикат может использоваться для быстрого завершения преобразования или для того, чтобы быстро занести весь вход в качестве значения некоторой переменной.

Шаблон удобно использовать при описании входных параметров вызываемой функции для того чтобы избежать их повторных проверок – ведь они были проверены перед вызовом функции. Подробности описаны в соответствующем разделе.

Важно также отметить, что шаблон _ не является множественным.

Примеры использования шаблона

Шаблон	Истина	Ложь
_	(всегда)	(никогда)

Операции SPARD и составные шаблоны

Составные шаблоны образуются при помощи применения **операций** (**конструкторов шаблонов**) SPARD к базовым и другим составным шаблонам. Такие шаблоны называются **операндами** шаблона или **подчинёнными** ему **шаблонами**. В результате подобного

применения возникают новые шаблоны с расширенной функциональностью. Поскольку составные шаблонов сами могут входить в состав других составных, становится возможно выразить сколь угодно сложные требования к входным данным.

Следует отметить, что подчинённые шаблоны могут совпасть со входными цепочками данных разной длины. При этом выбор итоговой длины цепочки, с которой совпал составной шаблон, определяется типом этого шаблона.

Далее описывается полный перечень операций SPARD.

Конкатенация (последовательность)

```
<Sequence> := <Template><Template>+
```

Определение

$$(a_1 a_2 \dots a_n)(x) :- x = x_1 x_2 \dots x_n, a_1(x_1), a_2(x_2), \dots a_n(x_n).$$

Основным способом объединения шаблонов является их конкатенация. В SPARD конкатенация никак не обозначается, шаблоны просто записываются последовательно друг за другом.

Предикат, соответствующий конкатенации, истинен тогда и только тогда, когда входной поток данных можно разбить на части (некоторые из них могут быть пустыми), число которых равно числу шаблонов в конкатенации, причём предикат каждого шаблона конкатенации истинен на соответствующей части входного потока.

Частным и достаточно распространённым случаем конкатенации является конкатенация символов – строка. Как следует из определения, предикат шаблона-строки истинен тогда и только тогда, когда входной текст в текущей позиции начинается с этой строки.

Строка также предоставляет удобный способ записи спецсимволов и пробелов в предикате. Вместо экранирования каждого из них одинарной кавычкой, можно записать всю строку в двойных кавычках, и в этом случае предикат будет требовать от начала входного текста точного соответствия этой строке.

Внутри двойных кавычек по-прежнему можно использовать одинарную для экранирования самих символов одинарной и двойной кавычки.

Примеры использования шаблона

Шаблон	Истина	Ложь
abc	abc	a
abc	abcdef	aabc
a.c	abc	ac
a'.p	aap	a\np
...	abc	ab
...	aaabbb	a
"a . "	a .b	a..
"abc'"def"	abc"def	"abc'"def"

Логические операции

Логическое И (&)

```
<And> := <Template> ('& <Template>)+
```

Определение

$$(A\&B)(x) :- A(x), B(x).$$

Предикат шаблона, заданного операцией `&`, истинен тогда и только тогда, когда все предикаты подчинённых шаблонов истинны в текущей позиции входных данных, причём все они **истинны на цепочке одной и той же длины** (то есть, на одной и той же цепочке). Запись шаблона представляет собой последовательную запись подчинённых шаблонов, разделённых символами `&`.

Примеры использования шаблона

Шаблон	Истина	Ложь
a&b	(никогда)	(всегда)
.&ab	(никогда)	(всегда)
.&a	a	b
a..&.b.&..c	abc	acb

Логическое ИЛИ (|)

```
<Or> := <Template> ('| <Template>)+
```

Определение

$$(A|B)(x) :- A(x);\ B(x).$$

Предикат шаблона, заданного операцией `|`, принимает значение ИСТИНА, если хотя бы один из предикатов операндов истинен в текущей позиции входных данных. Результирующей длиной совпавшей цепочки для шаблона является длина совпавшей цепочки для этого операнда. Запись шаблона представляет собой последовательную запись операндов, разделённых символами `|`.

Данный шаблон является множественным: каждый из операндов предоставляет отдельный вариант совпадения. **Варианты перебираются строго в порядке описания шаблонов**.

Примеры использования шаблона

Шаблон	Истина	Ложь
a|b	a	c
a|b	bcd	def
ab|x	x	b
a(b|x)	ax	bx

Логическое НЕ (!)

```
<Not> := '! <Template>
```

Определение

$$(!A)(x) :- A(x), fail.$$

Операция ! имеет ровно один подчинённый шаблон, записываемый справа от неё. Предикат шаблона отрицания истинен только тогда, когда предикат его подчинённого шаблона имеет значение ЛОЖЬ.

Важно отметить, что при совпадении шаблона отрицания неизвестно, на какой входной цепочке он выполнился (неизвестна её длина). К примеру, предикат шаблона !a истинен на входной строке “bcd”, но при этом шаблон может соответствовать как подстрокам “b”, “bc” и “bcd”, так и пустой подстроке входного текста. Поэтому считается, что предикат шаблона выполнился сразу на **всех** цепочках, начинающихся с текущего элемента. **Варианты совпадения перебираются в порядке возрастания длины**: сначала выбирается пустая цепочка, затем цепочка длиной 1 и т.д. до цепочки, равной всему оставшемуся входному потоку.

Для того чтобы уточнить длину цепочки, шаблон отрицания обычно связывают с другим шаблоном посредством операции &.

Обычная запись при использовании шаблона отрицания выглядит так:

1) .&!a – обозначает один совпавший символ;
2) (..)&!(a|bcd) – два совпавших символа и т.д.

Примеры использования шаблона

Шаблон	Истина	Ложь
!ab	b	a
.&!a	b	a
..&!(a|qw|bcd)	xy	qw
..&!(a|qw|bcd)	bc	b

Интервал (-)

```
<Interval> := <Template> '- <Template>
```

Определение

$$(A - B)(x): -x \geq A, x \leq B.$$

Интервал позволяет задавать нижнюю и верхнюю границу для входных данных. Для применения операции – во множестве входных данных должно существовать отношение линейного порядка. Если верхняя и нижняя граница представляют собой числа, то используется числовая функция сравнения («больше» или «меньше»). В противном случае используется лексикографическая функция сравнения («раньше» или «позже» в лексикографическом порядке). При лексикографическом сравнении максимальная длина совпавших входных данных не должна превышать длин нижней и верхней границ (если эти длины больше единицы, сопоставление будет множественным с перебором цепочек от большей к меньшей).

Интервал представляет собой удобный способ задавать множество последовательных элементов без необходимости перечислять их явно.

Примеры использования шаблона

Шаблона	Истина	Ложь
1-5	2	7
1-15	11	100
a-z	c	;
abc-def	bcdefg	h
abc-def	bcdefg	3
abc-def	b	k

Квантификаторы

Квантификаторы имеют в SPARD тот же смысл, что и в регулярных выражениях – они накладывают ограничения на число последовательных[2] сопоставлений подчинённого предиката. Квантификаторы всегда записываются справа от операндов.

[2] В виде конкатенации

Все квантификаторы являются множественными. Порядок перебора вариантов использует «жадный» алгоритм: сначала выбирается вариант совпадения с наибольшей возможной цепочкой, далее – цепочкой меньшей длины и т. д. Модификатор lazy меняет порядок перебора вариантов на противоположный.

Квантификатор «0 или 1» (?)

```
<ZeroOrOne> := <Template> '?
```

Операция ? делает необязательным выполнение предиката подчинённого шаблона. Таким образом, предикат составного шаблона всегда истинен. Разница между ситуациями, когда предикат подчинённого шаблона истинен или когда он ложен, заключается в длине совпавшей входной цепочки – во втором случае составной шаблон совпадает с пустой цепочкой.

Шаблон можно определить через другие шаблоны:

$$x? \equiv x|$$

где справа от | находится пустой шаблон.

Примеры использования шаблона

Шаблон	Истина	Ложь
a?	(всегда)	(никогда)
.?	(всегда)	(никогда)
(ab?c)?d	acd	cd

Квантификатор «1 и больше» (+)

```
<OneOrMore> := <Template> '+
```

Операция + допускает выполнение операнда произвольное число раз (большее нуля). Под произвольным числом раз имеется в виду то, что предикат шаблона-операнда должен принимать значение ИСТИНА на последовательных наборах входных данных, идущих без разрывов друг за другом.

Иными словами, шаблон можно формально определить как:

$$x+ \equiv \ldots|xxx|xx|x$$

где бесконечная запись математически понятна, но недопустима в синтаксисе SPARD. Можно описать шаблон корректно с точки зрения SPARD, используя рекурсию:

$$x+ \equiv x(x+|)$$

Примеры использования шаблона

Шаблон	Истина	Ложь
a+	aaa	baa
(a|b+c)+d	bbcabcd	bd

Квантификатор «0 и больше» (*)

```
<ZeroOrMore> := <Template> '*
```

Предикат шаблона, созданного с помощью операции *, истинен в тех же случаях, когда и предикат шаблона на основе +, но он также допускает совпадение с пустой цепочкой (при котором подчинённый предикат не выполняется ни разу):

x* ≡ x+|

Примеры использования шаблона

Шаблон	Истина	Ложь
a*	(всегда)	(никогда)
.*	(всегда)	(никогда)
(a|b*c)*d	bbcacd	bd

Квантификатор с шаблоном числа выполнений (#)

```
<Counter> := <Template> '# <Template>
```

Определение

$$(A\#B)(x) :- \big(AAA\ldots A\ (n \text{ раз})\big)(x), B(n).$$

Операция # позволяет задать шаблон для самого числа выполнений операнда. Повторяемый шаблон записывается слева от знака #, а шаблон числа выполнений – справа от него.

Примеры использования шаблона

Шаблон	Истина	Ложь
a#5	aaaaa	aaaab
a#2-4	aaaaaaaaa	a

Именованное множество (<>)

```
<Set> := '< <SetNameAndArgs> '>

<SetDefinition> := <SetNameAndArgs> <DefinitionSign> <Template>
<SetNameAndArgs> := '< <SetListBody> '>
<SetListBody> := <SetName> (', <SetArg> )*
<SetName> := <QName> & !<BuiltInSet>
<SetArg> := <InstructionVariable>
```

```
    <InstructionVariable> := '[ <Variable> ']

    <BuiltInSet> := BR | SP | s | t | d | i ; Возможно расширение
списка впоследствии

    <DefinitionSign> := ":="
```

Определение

$$< A > (x) : - Def(A, P), P(x).$$

где $Def(A, P)$, если существует определение $< A >:= P$.

Комбинируя шаблоны и операторы SPARD, можно получать достаточно сложные условия, налагаемые на входные данные. Для того чтобы не повторять в разных местах SPARD-файла описания одного и того же сложного шаблона, можно задать ему имя и ссылаться на него в других местах SPARD-описания. Именованный шаблон называется **именованным множеством**.

Подобное определение связано с тем, что предикат определяет множество цепочек, на которых он принимает значение ИСТИНА. С другой стороны, любое множество цепочек задаёт одновременно и предикат принадлежности к себе (характеристическую функцию). Поэтому понятия «предикат» и «множество» тождественны. В данном контексте множества имеют специально заданное имя, и на них благодаря этому можно ссылаться.

Далее в документе прилагательное «именованное» может опускаться; из контекста будет понятно, идёт ли о речь о множестве как о шаблоне SPARD или о множестве как математическом объекте.

Множества могут определяться рекурсивно через самих себя, что обеспечивает дополнительную гибкость при их использовании.

Определения множеств записываются вне правил преобразования SPARD, но могут перемежаться с ними и быть записаны в любом порядке (за исключением случая, описанного в конце данного раздела). Каждое определение, как и правило SPARD, располагается на отдельной строке. Простейшее определение множества имеет вид:

```
<A> := P
```

где A – имя множества, P – шаблон, определяющий элементы множества (те цепочки, на которых его предикат истинен). Примером может служить определение

арабских цифр (в язык встроено множество для цифр, но в целях демонстрации оно не применяется):

```
<Digit> := 0-9
```

На основании этого определения можно определить натуральные числа.

```
<Number> := (<Digit> & !0) <Digit>*
```

Видно, что множества значительно упрощают написание правил SPARD.

Множество может быть параметризовано. Определение множества с параметрами выглядит так:

$$< A, X_1, X_2 \dots, X_n >:= P$$

где A и P имеют тот же смысл, что и в простом определении множества, а X_1, X_2… X_n – имена параметров. Каждый параметр представляет собой либо шаблон α_k, либо уникальное для данного определения имя, заключённое в квадратные скобки $[N_k]$. Во втором случае шаблон P содержит этот же параметр. Например:

```
<Double, [A]> := [A][A]
```

Это множество требует наличия во входной цепочке двух следующих друг за другом одинаковых элементов.

При ссылке на множество можно указать конкретное значение параметра:

```
<Double, 1> => Yes
```

а можно указать переменную (рассматривается в разделе о переменных).

Параметризация множеств обеспечивает ещё более широкие возможности по повторному использованию шаблонов, нежели использование обычных множеств.

Вообще, параметры множества являются полноценными переменными в шаблоне определения. Подробнее про переменные можно прочитать в соответствующем разделе.

Для каждого множества допускается наличие в программе произвольного числа определений. Для проверки совпадения с входом будут использоваться все определения, подходящие по требуемому числу параметров (перебор осуществляется в порядке указания определений по аналогии с операцией ИЛИ). Это позволяет гибко указывать условия, налагаемые на элементы множества.

Примеры использования шаблона

Шаблон	Истина	Ложь
<A>:= 0|1|2 <A>	0	3
<A, [X]> := [X] <A, 1>	1	2

Предопределённые множества

Существуют множества, известные SPARD-преобразователю без стороннего определения. Использование этих множеств упрощает проведение специфических проверок входных данных. Такие множества можно рассматривать как дополнительные шаблоны, записываемые таким специфическим образом.

Имена предопределённых множеств зарезервированы и не могут использоваться для собственных множеств.

Предопределённые в SPARD множества

Множество	Описание	Определение
 	Перенос строки	\r\n|\r|\n
<SP>	Пробельный символ (без символов переноса строки)	зависит от системы
<s>	Строка текста	[line](.*)
<t>	Многострочный текст	.*
<d>	Цифра	0-9
<i>	Целое число	('+ | '- |)<d>+

Примеры использования шаблона

Шаблон	Истина	Ложь
1 2 3	1\r\n2\n3	123
1<SP>2<SP>+3	1 2\t 3	1\t2\r\n 3
a<s>b	afghb	afhdjf\nb
a<t>	adfdfd\nfddf	bgrrfjhf

Список (,)

```
<List> := <Template> (', <Template>)+
```

Определение

$$(A_1, A_2, \dots, A_n)(x) : -\ x\ =\ [x_1|\ x_2|\ \dots|\ x_n], A_1(x_1), A_2(x_2), \dots, A_n(x_n).$$

Список – это шаблон, имеющий неограниченное число операндов, разделяемых символами ,. Предикат списка принимает на входном потоке значение ИСТИНА, если

входной поток представляет собой также список с таким же количеством элементов, причём каждый элемент-предикат списка-предиката выполняется на соответствующем элементе входного списка.

В отличие от конкатенации, список не позволяет произвольным образом смещать границы совпадающих подцепочек входных данных – эти границы уже заранее размечены элементами списка. Поэтому в случае списка требования к входным данным строже.

Обычно список используется при описании формальных и фактических параметров функции, при описании параметризованных множеств и внутри инструкций.

Примеры использования шаблона

Шаблон	Истина	Ложь
a, b, c	a, b, c	a, bc
., ., x.*y	a, b, xy	a, b, y

Функция и вызов функции (@)

```
    <FunctionCall> := '@ <Template>

    <FunctionDefinition> := <FunctionName> <DefinitionSign> ( <Rule> | <Block> )
    <FunctionName> := <QName> & !<BuiltInFunction>
    <BuiltInFunction> := length | lower | upper | call ; Возможно расширение списка впоследствии

    <DefinitionSign> := ":="

    ; Блок определений
    <Block> := '( (<BR> <Rule>)* ')
```

Помимо предикатов, в SPARD используются и обычные функции. Если именованные множества позволяют повторно использовать один и тот же шаблон входа в разных местах правила и в разных правилах, функция даёт возможность повторно использовать результат преобразования. Кроме того, функция может использоваться внутри шаблонов, хотя сама она шаблоном и не является. Шаблон не может сгенерировать значение, он лишь проверяет входные данные. Но очень часто для проверки входных данных требуется произвести определённые вычисления. Эта задача и решается с помощью инструкций, а также функций.

Определение функции похоже на определение множества:

```
f := I => O
```

где `f` – имя функции, `I => O` – это стандартное правило преобразования SPARD с таким же предикатом и результатом, как и в глобальных правилах. В отличие от глобальных правил, это правило включается в работу только при вызове функции.

Пустой шаблон `I` описывает вход для функции, не имеющей входных параметров:

```
f := => O
```

Синтаксически правильно записывать и правило с пустым шаблоном результата, но практический смысл от него невелик:

```
f := =>
```

Описание функции может состоять из нескольких строк. Во всех строках используется одно и то же имя функции:

```
f := 1 => 2
f := 2 => 3
```

Существует и альтернативный способ определения функции, не требующий повторять её имя. В этом случае функция определяется сразу с помощью блока правил, заключённого в круглые скобки. Синтаксис подобного определения следующий:

```
f := (
   I1 => O1
   I2 => O2
   …
   In => On
)
```

Это определение является сокращённой альтернативой такому:

```
f := I1 => O1
f := I2 => O2
f := I3 => O3
```

Каждая функция в программе соответствует отдельному SPARD-преобразователю. Этот преобразователь называется **обобщённым**.

Обобщённый SPARD-преобразователь отличается от обычного тем, что может иметь несколько входных или выходных лент. Число лент соответствует числу входных и выходных параметров функции. Параметры функции разделены запятой, то есть содержатся в списке. Кроме того, в качестве шаблона входа такого преобразователя допустим пустой шаблон. Других отличий от обычного преобразователя нет.

Преобразователь, соответствующий функции, запускается в момент вызова функции, обозначаемом в SPARD-описании символом @.

Вызов функции – это не шаблон, но он подчинён всем правилам работы с шаблонами, содержит подчинённые шаблоны и находится в дереве выражений SPARD. Вызов функции имеет обязательный подчинённый шаблон – или имя вызываемой функции (в этом случае она вызывается без параметров), или список, содержащий имя функции и передаваемые ей входные параметры (они выступают в роли входных потоков для внутреннего SPARD-преобразователя).

Вызов функции считается успешным, если соответствующий ей преобразователь успешно прочитает все входные потоки до конца. Результатом вызова функции является набор частичных результатов, выданных преобразователем.

В случае неуспешного вызова функции программа генерирует исключение.

Так же, как и в случае с множествами, параметры функции представляют собой или шаблоны, или имена в квадратных скобках (переменные).

Вот пример описания функции склейки аргументов в обратном порядке и её вызова:

```
r := [A], [B], [C] => [C]@(r, [A], [B])
r := [A], [B]      => [B][A]
r := [A]           => [A]
_ => @(r, 14, 25, 36)
```

Вычисление итогового результата в данном примере происходит следующим образом.

1. Построение результата заключается в вызове функции r и выдаче её результата в выходной поток.

2. Перед вызовом функции вычисляются значения её аргументов. В данном случае эта операция тривиальна, так как аргументы функции фиксированы и равны «14», «25» и «36».

3. Вызывается функция r – активируется внутренний преобразователь, на вход которому поступает список из строк «14», «25» и «36».

4. В процессе проверки правил для функции выясняется, что сработало первое правило. Параметр A получает значение 1[3], параметр B – 2, параметр C – 3. Вычисляется результат, для получения которого снова вызывается r с параметрами «1» и «2».

5. Снова активируется внутренний преобразователь. Здесь важно отметить, что каждый вложенный (рекурсивный) вызов обрабатывается отдельным экземпляром (клоном) внутреннего преобразователя, благодаря чему исчезает риск появления побочных эффектов.

6. Для внутреннего преобразователя срабатывает второе правило, и он возвращает строку «21». Оба потока входа исчерпаны, преобразователь завершает свою работу.

7. Преобразователь уровнем выше вычисляет `[C]@(r, [A], [B])` и получает «321». Он возвращает частичный результат. Но входы ещё не полностью исчерпаны, поэтому вызов функции не завершён.

8. Аналогичная процедура производится для значений «4», «5» и «6». Возвращается строка «654».

9. Итоговый результат вызова функции – «321654». Он и становится результатом работы всей программы.

Данный пример демонстрирует механизм работы с функциями в SPARD. По своей сути они мало чем отличаются от функций в обычных языках программирования (за исключением своей рекурсивной структуры). Функции SPARD являются чистыми, то есть они никогда не изменяют входные данные и не зависят от внешних переменных. Благодаря такой особенности функций появляется возможность распараллеливать вычисления при выполнении SPARD-преобразований.

При вызове функции ей передаются данные, которые обычно уже были проверены другим шаблоном. Часто во вторичной проверке (шаблоном входа самой функции) нет необходимости. Для её устранения и для повышения производительности работы программы можно использовать шаблон _. Ниже приведён пример подобного использования:

```
1 => @(f, qwertyuiop)
f := .* => result (1)
f := _ => result (2)
```

[3] Почему именно 1, а не 14, объяснено в разделе про переменные.

Вариант (2) будет работать быстрее варианта (1), так как при вызове функции не будет производиться проверка входного параметра шаблоном `.*`.

Примеры использования функций

Определение	Вызов	Результат
`f := A => B`	`@(f, A)`	`B`
`f := [X] => [X][X]`	`@(f, A)`	`AA`
`f := [I],[J] => [I][J]`	`@(f, 1, 2)`	`12`
`f := => 200`	`@f`	`200`

Предопределённые функции

Как и в случае с множествами, в SPARD существуют предопределённые функции. Эти функции трудно определить с помощью синтаксиса SPARD, поэтому они предоставляют удобный способ нетривиальных (для SPARD) вычислений.

Имена предопределённых функций зарезервированы и не подлежат использованию для собственных функций.

Предопределённые в SPARD функции

Функция	Описание	Пример
`length`	Длина входного потока (число элементов в нём)	`@(length, aaa) = 3`
`lower`	Перевод входного текста в нижний регистр (для символьных преобразований)	`@(lower, Aaa) = aaa`
`upper`	Перевод входного текста в верхний регистр (для символьных преобразований)	`@(upper, Aaa) = AAA`
`call`	Применение функции, записанной в строке входных данных (возможность метапрограммирования)	`@(call, "a => b", aaaa) = bbbb`

Квалификаторы (\)

```
    <QName> := <BasicString> | <Qualifier>

    ; Квалификатор
    <Qualifier> := <BasicString> '\ <BasicString>
```

Квалификаторы применяются для уточнения имён используемых функций и множеств, определённых во внешних модулях. Квалификатор представляет собой обратную косую черту, разделяющую кодовое имя модуля, в котором объявлен объект (пространство имён) и имя самого объекта. Благодаря использованию квалификаторов имён удаётся однозначно идентифицировать используемый объект и избежать коллизий.

Инструкции ([])

```
    <Instruction> := '[ <InstructionTemplate> '] | '[
<InstructionTemplateWithArgument> '] <Template>
    <InstructionTemplate> := <Math> | <Comparison> | <Unification> |
<FunctionCall> | <Query> | <VariableSequence>
    <InstructionTemplateWithArgument> := <Modifier> |
<UnificationWithArgument>

    <Math> := <Add> | <Substract> | <Multiply> | <Divide> |
<Remainder>
    <Add> := <Template> ('+ <Template>)+
    <Substract> := <Template> '- <Template>
    <Multiply> := <Template> ('* <Template>)+
    <Divide> := <Template> '/ <Template>
    <Remainder> := <Template> '% <Template>
```

Инструкции – механизм расширения правил SPARD.

В процессе написаний правил SPARD очень часто возникают ситуации, когда шаблонов оказывается недостаточно для выражения всех требований к входным данным. Необходимо произвести какие-то вычисления с данными и затем проверить результат этих вычислений. Также часто возникает необходимость передать какие-то данные из входа в результат. Иногда необходимо задать модификаторы преобразования, изменяющие механизм работы некоторых шаблонов. Для решения всех этих задач и существуют инструкции.

Всё, что невозможно описать с помощью шаблонов, следует описывать с помощью инструкций. Инструкции используются в дереве шаблонов наравне с другими шаблонами, но функционируют иначе. Многие инструкции являются полноценными шаблонами.

Каждая инструкция заключается в квадратные скобки ([]). Внутри скобок действует свой синтаксис, который оптимизирован под выполнение вычислений. В частности, символы > и < используются в качестве операторов «больше» и «меньше» соответственно, а не в качестве границ множеств.

Ниже расписаны все существующие в текущей версии языка инструкции с примерами их использования.

Некоторые инструкции, помимо аргументов, заключённых внутри квадратных скобок, требуют наличия **дополнительного аргумента** справа от инструкции. В каждом случае это оговаривается отдельно.

Контекст преобразования

Контекст преобразования – это область памяти, разделяемая шаблоном входа и шаблоном результата. Благодаря наличию контекста шаблон входа может передать часть входных данных в шаблон результата (напрямую передать невозможно). Передача данных осуществляется через **переменные** – именованные области контекста. Каждая переменная имеет уникальное имя, которое **должно начинаться с прописной буквы**.

Если внутри инструкции требуется написать прописную букву, не обозначающую имя переменной, это буква экранируется апострофом '.

Шаблон входа и шаблон результата могут использовать переменные и для собственных нужд, объявив переменную в одном месте и использовав её в другом.

Контекст преобразования был неявно определён в начале спецификации, когда были введены параметры предиката и результата. Контекст как раз и представляет собой способ передачи значений этих параметров.

Унификация (=)

```
<Unification> := <Template> '= <Template>
<UnificationWithArgument> := <Template> '=
```

Унификация SPARD работает практически так же, как и унификация в языке Пролог. Основной результат выполнения унификации – связывание значений двух переменных, переменной и значения или просто сравнение двух значений. Результат унификации принимает значение ИСТИНА или ЛОЖЬ и также используется для проверки совпадения шаблона со входом. Вот пример унификации переменных:

```
[X = Y]
```

После выполнения такой унификации любое изменение значения переменной X будет вести к такому же изменению значения переменной Y (и наоборот).

Если в момент унификации обе переменные уже имели значения и эти значения различаются, унификация возвращает значение ЛОЖЬ. Во всех остальных случаях унификация проходит успешно.

Унификация переменной и значения происходит так:

```
[X = R]
```

где R – результат, вычисляемый в момент выполнения унификации. R может представлять собой выражение, где допускаются арифметические операции, или вызов функции. На данный момент поддерживаются операции +, -, *, / и % (получение остатка от деления). Например,

```
[X = 2 + 5 * 8]
```

или

```
[X = Y + 1]
```

Такая унификация завершается успешно, если переменная не имела значения ранее или если её значение совпало с результатом вычисления R. В результате выполнения унификации переменная при отсутствии значения получает его. Как и во всех декларативных языках программирования, переменная, получившая значение, больше не может его изменить.

Унификация может производиться и вовсе без одиночных переменных слева и справа. В этом случае значения левой и правой частей сравниваются друг с другом. Результат сравнения и становится результатом унификации:

```
[A + 2 = B - 3]
```

Ещё один вариант унификации требует наличие дополнительного аргумента для шаблона-инструкции.

```
[X=]P
```

где P – шаблон, сопоставляемый с входными данными. В случае успеха сопоставления в переменную X заносится совпавшая с шаблоном цепочка входных данных. Подобная унификация – способ получить в качестве значения переменной элементы входных данных. Если же переменная X на момент проверки шаблона уже

имела какое-то значение, произойдёт унификация значений переменной и совпавших данных.

Пример:

```
[X=](a.b)
```

При проверке входного текста “axbyc” предикат шаблона будет выполнен, а переменная X примет значение “axb”, которое затем можно использовать (например, вывести в результат).

С помощью унификации становится возможным описать более сложные множества и функции. Например:

```
<DividedBy, [X]> := [N=]_[0 = N % X]
```

– это множество описывает числа, которые делятся на [X]. К примеру, множество <DividedBy, 2> описывает чётные числа.

Вот пример описания функции суммирования, входом для которой должен являться список из двух аргументов:

```
sum := [A], [B] => [A + B]
=> @(sum, 4, 5)
```

При вызове функции в качестве фактического параметра ей передаётся список также из двух элементов – 4 и 5 (имя функции перед её вызовом из списка исключается).

Примеры использования шаблона (зависят от реализации SPARD-преобразователя)

Шаблон	Результат
[A = B] B = 2	ИСТИНА A = 2
[A = B] A = 1, B = 2	ЛОЖЬ
[A = B]	ИСТИНА A = B
[A = @(length, aaa)]	ИСТИНА A = 3
[A=](.&!b)* вход: aabbcc	ИСТИНА A = aa

Использование переменной

```
<Query> := <Variable> (', <Variable>)*
<Variable> := (A-Z) <BasicString>
```

Для использования переменной достаточно указать её имя в квадратных скобках. Если переменная используется в шаблоне входа, то она выступает также как шаблон. Если переменная имеет значение, это значение сравнивается с входом.

Если же значение переменной не задано, производится неявная унификация.

Если переменная встречается в шаблоне результата, её значение просто выводится в результат. В случае отсутствия у переменной значения в результат ничего не выводится.

Если значением переменной является объект, то можно запросить вместо самого объекта какое-либо из его свойств. Значением свойства снова может быть объект; поэтому в качестве запроса к контексту может использоваться цепочка из имён, разделяемый запятыми: первое имя обозначает имя переменной, следующее имя – имя свойства объекта – значения переменной, затем – имя свойства у значения свойства и т.д.

Например, запрос может выглядеть следующим образом:

```
[X, FullName, Surname]
```

Если значением переменной `X` является объект `{FullName: {Name: Вася, Surname: Иванов}}`, то результатом выполнения запроса будет «Иванов».

Расширенный синтаксис запросов обычно используется при извлечении из входа найденных совпадений.

Примеры использования шаблона

Шаблон	Истина	Ложь
`[A=]x[A]`	`xx`	`xy`
`[A=].* [A=B][B]`	`xy\rxy`	`xy\ryx`

Неявная унификация

Если в шаблоне входа встречается переменная, не имеющая значения, она автоматически унифицируется с одним элементом из входного потока данных. К примеру, преобразование

```
[X] => [X][X]
```

удваивает каждый входной элемент, при этом шаблон входа явно нигде не задаётся.

Данная неявная унификация используется исключительно для краткости записи кода. Полная запись преобразования выше выглядит так:

```
[X=](.|{}) => [X][X]
```

Неявно заданные переменные

Шаблон результата позволяет использовать переменные, не определённые во входе. Эти переменные имеют специальную форму записи: знак `$` и номер (начиная с нуля). При успешном совпадении шаблона входа с входными данными такие переменные принимают значения соответствующих аргументов входа (переменная `$0` становится равной первому входному аргументу преобразователя, `$1` – второму и т.д.). Данная запись позволяет избавиться от излишних описаний унификаций.

Проще всего понять смысл таких переменных на примере. Запись:

```
add := _, _ => [$0 + $1]
```

является сокращённым вариантом записи:

```
add := [A=]_, [B=]_ => [A + B]
```

При использовании таких неявно заданных переменных код становится короче и нагляднее.

Неявные аргументы функции

Использование неявно заданных переменных позволяет сократить код ещё одним образом: можно не указывать аргументы функции, совпадающие с такими переменными. SPARD-преобразователь из анализа правой части функции самостоятельно определит число её входных аргументов. Функцию `add` из предыдущего раздела можно записать и так:

```
add := => [$0 + $1]
```

Последовательности переменных (:)

```
; Последовательность переменных
<VariableSequence> := <Variable> (': <Variable>)*
```

Вместо одной переменной в квадратных скобках можно записывать последовательности переменных, разделяя их имена двоеточием (`:`). Данная последовательность функционирует как неявная унификация: первая указанная переменная унифицируется с первым элементом входных данных, вторая – со вторым и т.д. Последняя переменная последовательности унифицируется уже не с единичным элементом, а со всей оставшейся частью («хвостом») входного потока данных.

Для успешного сопоставления с данным шаблоном последняя переменная должна унифицироваться по крайней мере с одним элементом. Унификация последней переменной с пустой цепочкой не производится.

Данная запись служит исключительно для удобства и сокращения длины программы.

Шаблон можно использовать и в результате. В этом случае он последовательно возвращает значения переменных, входящих в него.

Примеры использования шаблона

Шаблон	Вход	Значения переменных
[H:T]	abcd	H = a, T = bcd
[A:B:C]	abcd	A = a, B = b, C = cd
[A:A:T]	abcd	ЛОЖЬ

Инструкции-операторы

```
    <Operator> := <Modifier> | <Foreach> | <Matches> | cont | one | debug | cache
```

Инструкции-операторы – это строки или списки из строк, записываемые в квадратных скобках. Они предписывают выполнить какое-то действие в данный момент преобразования или влияют на дальнейший процесс преобразования.

Операторы – модификаторы преобразования (on и off)

```
    <Modifier> := (on | off |) <Mode>
    <Mode> := ignoresp | keepinitiator | left | m | match | mvar | multi | lazy | line | lrec | opt | ci
```

Некоторые шаблоны могут работать в разных режимах. Чтобы изменить стандартное поведение шаблона, достаточно задать перед ним соответствующий список из оператора с модификатором в квадратных скобках. Оператор `on` включает модификатор, `off` – выключает его. Оператор `on` при записи можно опускать (записывая просто имя модификатора). По умолчанию все модификаторы выключены. Ниже дано описание всех модификаторов в текущей версии SPARD.

Модификаторы SPARD

Модификатор	Применяется к	Описание
ignoresp	Строка	При сравнении строки с входом несовпадающие пробельные

		символы во входе пропускаются
keepinitiator	Шаблон	В настоящий момент используется в служебных целях
left	@	Функция вызывается «справа налево» (выполняется обратное преобразование)
lazy	*, +, ?	Выполняются ленивые сравнения (в качестве первого подходящего варианта выбирается цепочка наименьшей длины и далее по увеличению длины)
m	<S>	Часть входа, совпавшая с множеством, сохраняется при завершении проверки правила. Используется для извлечения данных из входного потока
mvar	[X=]P	Значение переменной сохраняется при завершении применения правила. Используется для извлечения данных из входного потока
match	Шаблон	То же, что и m, но при этом ещё и наследуется при проверке вложенных множеств. Использование этого модификатора позволяет извлечь из входа дерево значений (произвести синтаксический анализ)
multi	@	Вызов функции является многозначным
line	., ^, $	«Любой символ» включает в себя также и символы переноса строки
lrec	Шаблон	«Подсказывает» преобразователю, что шаблон содержит левую рекурсию
opt	:	Если подобное свойство у объекта отсутствует или не задано, то оно не проверяется шаблоном свойства
ci	Строка	Нечувствительность к регистру для символьных преобразований

Если модификатор применяется к некоторому шаблону, то он автоматически применяется ко всем внутренним шаблонам данного. Таким образом, каждый модификатор сразу применяется к дереву выражений.

При переходе проверки шаблона внутрь определения множества наследуются лишь модификаторы ignoresp, match, ci и lrec.

Примеры использования модификаторов

```
[on, lazy](.*)
[ignoresp](..a.[off, ignoresp](b.+c))
```

Механизм совпадений

Часто возникает необходимость проверить входные данные с помощью именованного множества, имеющего сложную иерархическую структуру (его определение содержит другие множества и т.д.). При этом также обычно имеется необходимость сохранить часть совпавшего входа и использовать её при построении результата. При проверке входа обычным шаблоном это задача решается с помощью переменных. Но при проверке входа множеством доступ к внутренним переменным шаблона-определения отсутствует (это согласуется с функциональной чистотой языка). Чтобы извлечь из входа, совпавшего с множеством, интересующую часть, можно использовать параметры множества, но иногда это бывает неудобно (параметров может потребоваться слишком много). Иногда возникает задача построить дерево значений, соответствующее иерархии множеств шаблона (произвести синтаксический анализ). Для этих целей в SPARD существует механизм совпадений.

Суть механизма совпадений заключается в том, что при проверке шаблона внутри множества мы может сохранить часть входа и «вернуть» его «наверх», в «вызывающий» данную проверку множества шаблон. «Вызывающий» шаблон затем сможет получить доступ к сохранённому совпадению посредством запроса к контексту.

В качестве совпадения допускается сохранять либо значение переменной, либо часть входа, совпавшую с некоторым множеством. Для функционирования механизма совпадений существуют модификаторы `mvar`, `m` и `match`.

Модификатор mvar

Пусть мы хотим извлечь из входных данных информацию о возрасте человека, а на вход подана строка «Имя: Иван, Фамилия: Иванов, Отчество: Иванович, Возраст: 45»:

```
<Person> := <Name><Sep><Surname><Sep><SecondName><Sep><Age>
<Sep> := ',<SP>*
<Name> := <Pair, Имя>
<Surname> := <Pair, Фамилия>
<SecondName> := <Pair, Отчество>
<Age> := <Pair, Возраст>
<Pair, [N]> := [N] ':<SP>*.+
```

Задачу можно решить, передавая параметры для множеств:

```
<Person, [Age]> := <Name><Sep><Surname><Sep><SecondName><Sep><Age, [Age]>
<Age, [Age]> := <Pair, Возраст, [Age]>
```

```
<Pair, [N], [V]> := [N] ':<SP>*[V=](.+)
```

В этом случае переменная будет доступна под именем `Age`.

Механизм совпадений позволяет решить проблему более изящно, не передавая параметры вглубь определений множеств. Кроме того, когда извлекается значительный объём различных данных, количество передаваемых параметров становится слишком велико.

Для работы механизма совпадений достаточно задать модификатор `[mvar]` перед переменной:

```
<Pair, [N]> := [N] ':<SP>*[mvar][V=](.+)
<Age> := <Pair, Возраст>
```

При этом в случае совпадения шаблона `<Person>` совпавшее значение возраста можно получить с использованием оператора `matches`:

```
[matches, Person, Age, Pair, V]
```

или (оператор можно опустить):

```
[Person, Age, Pair, V]
```

где список имён описывает путь к переменной (включает имена множеств), содержащей совпавшую часть образца.

Модификатор m

Модификатор `m` работает похожим на `mvar` образом. Отличие заключается в том, что `m` сохраняет в качестве совпадения часть входа, совпавшую с некотором множеством, а не с переменной. Его удобно использовать при проверке входа множеством и при отсутствии необходимости создавать лишнюю переменную.

Предыдущий пример может быть записан иначе:

```
<Person> := <Name><Sep><Surname><Sep><SecondName><Sep>[m]<Age>
```

Тогда полученное совпадение будет доступно по имени:

```
[matches, Person, Age]
```

или просто

```
[Person, Age]
```

Модификатор match

Модификатор `match` расширяет возможности модификатора `m` и позволяет создать дерево совпадений по входу. Узлами дерева становятся отрезки входа, совпавшие с

отдельными множествами. Узлы подчиняются друг другу в соответствии с вхождениями множеств друг в друга. В результате с помощью указания одного модификатора можно произвести синтаксический анализ всего входа и затем обрабатывать его как разобранное дерево значений.

Предыдущий пример может быть переписан следующим образом:

```
[match]<Person> => [Person, Age]
```

Это правило легко извлекает из входа возраст человека. Никакие модификаторы внутри определения множества не нужны.

Оператор matches

```
<Matches> := matches', <Query>
```

Оператор `matches` позволяет осуществить запрос к найденным во входе совпадениям, которые были сохранены в контексте посредством модификаторов `m`, `mvar` и `match`. Принцип его работы аналогичен принципу получения значения переменной. Оператор `matches` можно опускать, если имя совпадения не совпадает с именем переменной в текущем контексте.

Оператор cont

Оператор `cont` задаёт контекстно-зависимый участок правила (в терминах формальных грамматик). Шаблон, помеченный таким оператором, при совпадении с входным потоком не сдвигает его на размер совпавшей части, как это обычно происходит. Входной поток не сдвигается вообще.

Этот модификатор удобно использовать для проверки наличия в последующей цепочке входного потока каких-то элементов без сдвига входа. Такой подход позволяет описывать в SPARD контекстно-зависимые правила.

Допустим, входной поток представляет собой последовательность чисел. Необходимо в выходной поток вывести сумму первого и второго числа, второго и третьего и т.д. Без использования контекстного правила такую задачу выполнить трудно: если написать правило для сложения двух чисел, вход сразу передвинется на два числа, и не будет возможности вернуть его назад для суммирования второго числа с третьим. Проблема решается с помощью модификатора `cont`:

```
{[N=]_}[cont]{[M=]_} => {[N + M]}
```

Контекстный модификатор разрешено использовать где угодно, но практический смысл он имеет только в конце шаблона входа.

Оператор one

Оператор one – это классическое отсечение в логических языках программирования. Он позволяет своему подчинённому шаблону один раз проверить входные данные, но при повторном обращении всегда возвращает ЛОЖЬ. Этот оператор удобен в тех ситуациях, когда подчинённый шаблон имеет тенденцию перебирать большое число ненужных совпадений (например, шаблон `.*` при условии, что он должен совпасть только со строкой максимальной длины).

Примеры использования шаблона

Шаблон	Истина	Ложь
`([one].*)a`	`(никогда)`	`aaa`
`([one]a?)ab`	`aab`	`ab`

Оператор foreach

```
<Foreach> := foreach', <InstructionVariable>, <FunctionCall>
```

Оператор `foreach` предоставляет возможность перебирать значения, выдаваемые некоторой функцией. Это бывает удобно, когда в момент вызова функции ещё окончательно неизвестно, какой именно результат она должна возвратить. С использованием оператора `foreach` можно использовать сначала первый результат работы функции, а в случае невозможности доказательства предиката – вернуться и попробовать другой вариант. Примером может служить функция, возвращающая начальную форму слова по его словоформе. Если производится разбор предложения на естественном языке, сначала выбирается одна гипотеза относительно начальной формы каждого слова в предложении, а в случае невозможности распознать структуру предложения гипотеза меняется. Так происходит до тех пор, пока не будет найдена удачная комбинация начальных форм для каждого слова, удовлетворяющая правилам синтаксиса.

Форма записи предиката:

$$[foreach, X, @(f, a_1, a_2, \dots, a_n)]$$

где X – имя переменной, в которую помещается результат каждого вызова функции, @ – вызов функции (возможно, с параметрами).

Оператор debug

Данный оператор никак не влияет на результат преобразования. Его единственное назначение – указать точку останова при отладке программы. Реализация SPARD, дошедшая до данного оператора, должна остановиться и предоставить программисту возможность изучить текущее состояние преобразования (включая инспектирование значений переменных). После этого программист может продолжить преобразование, и управление перейдёт к подчинённому для данного оператора шаблону.

При запуске преобразователя без режима отладки оператор никак себя не проявляет.

Оператор cache

Оператор `cache` привносит в SPARD мемоизацию – возможность запомнить результаты сравнения образца с шаблоном для заданной позиции в образце и для заданного шаблона. Данное запоминание (кэширование) ускоряет преобразование, если проверка на совпадение с шаблоном занимает слишком много времени. Использование данного оператора позволяет сократить временную сложность алгоритма преобразования с экспоненциальной до полиномиальной.

В то же время важно помнить, что мемоизация требует серьёзных затрат на хранение запомненных результатов, поэтому она не включается по умолчанию. Пользователь должен самостоятельно выявить проблемные места в своих программах и снабдить их оператором `cache`.

На результат преобразования этот оператор никак не влияет.

Допустим, что входной текст следует разбить на несколько подстрок, разделённых специфическими символами, причём желательно выбрать подстроки наибольшей длины:

```
<s> ";" <s> "-" <s> "(" <s> ")"
```

Можно убедиться, что для входного текста вида “…;…-…(…)…;…;…;…;…-…(…” (то есть имеющего большое число ложных совпадений) преобразование будет выполняться крайне медленно. При включении кэширования результатов оставшейся части шаблона скорость преобразования резко возрастает:

```
<s> [cache](";" <s> [cache]("-" <s> [cache]("(" <s> ")" )))
```

Предикаты (внутри инструкций)

```
<Comparison> := <Template> ('> | '< | "!=") <Template>
```

На данный момент поддерживаются предикаты больше (>), меньше (<) и не равно (!=). Условия, при которых они выполняются, очевидны. В данном варианте инструкция вычисляется как обычный предикат SPARD.

Вот пример использования такой инструкции в шаблоне, допускающем входные строки длиной более 10 символов.

```
[X=]_[@(length, X) > 10]
```

Примеры использования шаблона

Шаблон	Истина	Ложь
[A=].+[A < 5]	3	6
[A=].[B=].[A > B]	64	7a
[A=].[B=].[A != B]	57	44

Свойство (:)

```
<Property> := <BasicString> ': <Template>
```

Операция : используется для указания значения свойства при описании объекта. Этот оператор используется лишь внутри оператора {}.

Предикат оператора принимает значение ИСТИНА только тогда, когда входной объект обладает требуемым свойством и шаблон свойства совпадает со значением свойства объекта.

Следует отметить, что имя свойства – это хоть и не шаблон, но оно записывается по правилам построения шаблонов и может не быть постоянным выражением. Оно может содержать те же параметры, что и шаблон результата правила, а в момент проверки предиката шаблона входа на истинность вместо этих параметров подставляются конкретные значения.

Поведение шаблона может быть изменено при помощи модификатора opt. В этом режиме предикат шаблона возвращает также значение ИСТИНА, если у объекта требуемое свойство отсутствует.

Примеры использования шаблона (зависят от реализации SPARD-преобразователя)

Шаблон	Истина	Ложь
A:B	new {A="B"}	new {A="C"}
A:.	new {A="B"}	new {}
[X='A][Y='B][X]:[Y]	new {A="B"}	new {A="C"}

Объект ({})

```
<Object> := '{ <ObjectBody> '}
<ObjectBody> := <BasicString> | <BasicString>? (', <Property>)+
```

Для нетекстовых входных данных используется операция {} – фигурные скобки, внутри которых описывается объект.

Объект в SPARD – это некоторая абстрактная сущность, имеющая имя и набор свойств. Допускается использование объектов, имеющих только имя или только набор свойств. Имя объекта (если оно есть) всегда записывается в качестве первого подчинённого шаблона, свойства следуют далее. Все подчинённые шаблоны разделены запятыми, то есть представляют собой обычный список SPARD.

Определение истинности для предиката шаблона объекта происходит следующим образом. Конкретная реализация SPARD-преобразователя должна уметь определять имя для входного объекта и значение свойства по заданному имени свойства. При наличии подобной функциональности преобразователя предикат шаблона принимает значение ИСТИНА, если имя шаблона (само являясь шаблоном) совпадает с именем объекта, а значения свойств (также шаблоны) совпадают со значениями соответствующих свойств входного объекта. Если входной объект не обладает одним из требуемых свойств или значение свойства пусто, предикат всего шаблона принимает на этом объекте значение ЛОЖЬ (это поведение можно изменить при помощи модификатора opt).

Шаблон {} обозначает один элемент во входном потоке данных. Но с его помощью можно легко описать классическое разделение последовательности всех входных данных на голову и хвост:

```
[H=]{}[T=]_
```

Примеры использования шаблона (формы объектов зависят от реализации SPARD-преобразователя)

Шаблон	Истина	Ложь
{0}	0	1
{0}{1}{2}	0,1,2	2,3,4
{Info,Name:John}	class X {Type:Info,	class X {Type:Info,

	Name: John}	Name:Adam}
{Info,Name:John}	new {Type:Info, Name: John}	class X {Type:Info}
{Info,Name:John}	Info:John	3
{A:B, C:D, E:F}	{B, D, F}	{A, C, E}

Многострочные шаблоны

Шаблон можно разместить на нескольких строках, если он слишком громоздкий. Подобное размещение позволяет повысить читабельность кода.

В записи шаблона допускается переход на новую строку (который игнорируется) при условии, что этот переход находится внутри круглых скобок. Таким образом, простейший способ записать многострочный шаблон – это заключить его целиком в скобки.

Следующие шаблоны являются многострочными:

```
<A> := abc(
def |
ghi)

f := (abc
=> d
)

(abcdefshsgishigigfifhighgf
dfefejfioejfejrfejfpeofjepo
fefefhefiefhifhefhfe) => x
```

Единственным исключением из описанного правила является запись

```
f := (
a => b
c => d
)
```

Определение функции, состоящее только из открывающей скобки на первой строке, обрабатывается особым образом. Данный синтаксис обозначает, что тело функции состоит из блока правил, каждое из которых расположено на отдельной строке. Внутри этих правил многострочные шаблоны также возможны (но нежелательны в силу ухудшения читабельности):

```
f := (
   (a =>
      b)

   c => (
      d
```

```
    )
)
```

Глобальные инструкции

```
; Глобальная инструкция
<GlobalInstruction> := '[ <GlobalInstructionBody> ']
<GlobalInstructionBody> := <Module>
```

Глобальные инструкции используются для изменения принципа работы всего преобразователя, а не отдельных его частей. В настоящий момент поддерживаются только инструкции подключаемых внешних модулей.

Внешний модуль

```
; Внешний модуль
<Module> := module ', (.&!',)+ ', <BasicString>
```

SPARD-программа может располагаться в нескольких файлах. Файл, содержащий описание основного преобразования, считается главным. Остальные файлы могут содержать только определения и также ссылки на модули.

Ссылки из основного файла на остальные оформляются в виде глобальных инструкций. Каждая ссылка содержит слово `module`, название файла и кодовое имя (псевдоним), под которым будут доступны все множества и функции, описанные в нём. Одноимённые объекты, описанные в разных файлах, считаются различными.

Вот пример подключения внешнего модуля:

```
[module, other.spard, o]
```

Для того чтобы обратиться к объектам из файла `other.spard`, нужно предварить имена этих объектов квалификатором с ключом `o`. Если, предположим, в этом файле описаны функция `f` и множество `A`, то из основного файла к ним можно обратиться так:

```
@(o\f, arg1, arg2,…)
<o\A, arg1, arg2…>
```

Однако если в основном файле не заданы объекты с такими именами, квалификатор можно опустить. Если же при этом одноимённые объекты будут иметься и в других модулях, вызываться будут объекты из модуля, подключённого первым.

Обратимые преобразования

Синтаксис SPARD позволяет описывать преобразования, которые могут быть обратимы. Примером может служить пословный перевод с одного языка на другой. Обратное преобразование должно осуществить обратный перевод.

```
стол = table
книга = book
ручка = pen
```

В обратимом преобразовании направление применение каждого правила задаётся операцией =, а не =>. Реализация SPARD-преобразователя должна позволять указывать, в каком направлении будет применяться преобразование. В зависимости от выбранного направления одна из частей правил становится шаблоном входа, а другая – шаблоном результата.

Сводная информация

Глоссарий

SPARD – язык описания преобразований потоковых данных. Язык, предназначенный для удобного описания преобразований потоков данных в другие потоки данных.

SPARD-преобразователь – абстрактное вычислительное устройство, содержащее входную и выходную ленты и осуществляющее преобразование входных данных в выходные согласно заданной программе на SPARD.

Поток данных – последовательность однородных данных, доступных строго в порядке их следования без возможности возврата (для осуществления механизма возврата внутри преобразователя применяется кэширование).

Правило преобразования – часть преобразующей функции, задающая соответствие между подмножеством входных данных и результатом, сопоставляемым этому подмножеству. Правило состоит из шаблона входа, шаблона результата и направления применения.

Шаблон данных – запись, которая после выполнения корректных подстановок в неё становится цепочкой потоковых данных. Все данные, которые можно получить в результате корректных подстановок в шаблон, называются **допускаемыми** этим шаблоном (или **удовлетворяющими** ему, **совпадающие** с ним). Корректные подстановки определяются синтаксисом языка SPARD.

Таким образом, каждый шаблон задаёт **множество** всех данных, которые удовлетворяют ему. Одновременно с этим определяется и **предикат**, который истинен только на данных из этого множества. Шаблон можно также рассматривать и как **выражение**, представляющее собой дерево из простейших шаблонов.

Семантика шаблона SPARD может быть определена:

а) указанием корректных подстановок для него;

б) перечислением всех элементов соответствующего множества;

в) определением соответствующего предиката.

Синтаксис шаблона SPARD определяется как синтаксис соответствующего выражения.

Полное определение синтаксиса SPARD на самом SPARD

```
    ; Программа
    <Program> := (<SPARDLine> <Comment>? <BR>)*
    <SPARDLine> := <Definition> | <Rule> | <GlobalInstruction> |
    <Definition> := <SetDefinition> | <FunctionDefinition>

    ; Комментарий
    <Comment> := '; <String>[line]$

    ; Правило
    <Rule> := <Template> <Direction> <Template>
    <Direction> := "=>" | '= | "<="

    ; Определение именованного множества
    <SetDefinition> := <SetNameAndArgs> <DefinitionSign> <Template>
    <SetNameAndArgs> := '< <SetListBody> '>
    <SetListBody> := <SetName> (', <SetArg> )*
    <SetName> := <QName> & !<BuiltInSet>
    <SetArg> := <InstructionVariable>

    <QName> := <BasicString> | <Qualifier>

    ; Квалификатор
    <Qualifier> := <BasicString> '\ <BasicString>

    <InstructionVariable> := '[ <Variable> ']

    ; Встроенные множества
    <BuiltInSet> := BR | SP | s | t | d | i ; Возможно расширение
списка впоследствии

    ; Определение функции
    <FunctionDefinition> := <FunctionName> <DefinitionSign> ( <Rule> |
<Block> )
    <FunctionName> := <QName> & !<BuiltInFunction>

    ; Встроенные функции
    <BuiltInFunction> := length | lower | upper | call ; Возможно
расширение списка впоследствии

    ; Знак определения
    <DefinitionSign> := ":="

    ; Блок определений
    <Block> := '( (<BR> <Rule>)* ')
```

```
    ; Глобальная инструкция
    <GlobalInstruction> := '[ <GlobalInstructionBody> ']
    <GlobalInstructionBody> := <Module>

    ; Внешний модуль
    <Module> := module ', (.&!',)+ ', <BasicString>

    ; Шаблон
    <Template> := <Primitive> | <Single> | <Binary> | <Polynomial> |
'( <Template> ')

    ; Простейший шаблон
    <Primitive> := <Empty> | <OpenLine> | <CloseLine> | <Any> |
<Anything> | <BasicString>

    ; Пустой шаблон
    <Empty> :=

    ; Начало потока
    <OpenLine> := '^

    ; Конец потока
    <CloseLine> := '$

    ; Любой символ
    <Any> := '.

    ; “Хвост” потока
    <Anything> := '_

    ; Шаблон с одним параметром
    <Single> := <Unary> | <Dual>

    ; Унарная операция, расположенная рядом с аргументом
    <Unary> := <FunctionCall> | <Not> | <ZeroOrOne> | <ZeroOrMore> |
<OneOrMore>

    ; Вызов функции
    <FunctionCall> := '@ <Template>

    ; Логическое отрицание
    <Not> := '! <Template>

    ; Квантор: 0 или 1
    <ZeroOrOne> := <Template> '?

    ; Квантор: от 0 и больше
    <ZeroOrMore> := <Template> '*

    ; Квантор: от 1 и больше
    <OneOrMore> := <Template> '+
```

```
    ; Унарная операция, обозначаемая скобками
    <Dual> := <Instruction> | <Object> | <Set>

    ; Инструкция
    <Instruction>    :=    '[    <InstructionTemplate>    ']    |    '[
<InstructionTemplateWithArgument> '] <Template>

    ; Объект
    <Object> := '{ <ObjectBody> '}
    <ObjectBody> := <BasicString> | <BasicString>? (', <Property>)+

    ; Свойство объекта
    <Property> := <BasicString> ': <Template>

    ; Именованное множество
    <Set> := '< <SetNameAndArgs> '>

    ; Бинарный шаблон
    <Binary> := <Counter> | <Interval>

    ; Квантор: число возможных использований шаблона задаётся другим
шаблоном
    <Counter> := <Template> '# <Template>

    ; Интервал
    <Interval> := <Template> '- <Template>

    ; Шаблон с неограниченным числом аргументов
    <Polynomial> := <Sequence> | <And> | <Or> | <List>

    ; Конкатенация
    <Sequence> := <Template><Template>+

    ; Логическое И
    <And> := <Template> ('& <Template>)+

    ; Логическое ИЛИ
    <Or> := <Template> ('| <Template>)+

    ; Список шаблонов
    <List> := <Template> (', <Template>)+

    ; Шаблон внутри инструкции
    <InstructionTemplate> := <Math> | <Comparison> | <Unification> |
<FunctionCall> | <Query> | <VariableSequence>

    ; Арифметическая операция
    <Math>  :=  <Add>  |  <Substract>  |  <Multiply>  |  <Divide>  |
<Remainder>
```

```
    ; Сложение
    <Add> := <Template> ('+ <Template>)+

    ; Вычитание
    <Substract> := <Template> '- <Template>

    ; Умножение
    <Multiply> := <Template> ('* <Template>)+

    ; Деление
    <Divide> := <Template> '/ <Template>

    ; Остаток от деления
    <Remainder> := <Template> '% <Template>

    ; Сравнение
    <Comparison> := <Template> ('> | '< | "!=") <Template>

    ; Унификация
    <Unification> := <Template> '= <Template>

    ; Запрос к контексту
    <Query> := <Variable> (', <Variable>)*

    ; Переменная
    <Variable> := (A-Z) <BasicString> ; Допускаются прописные буквы
любого языка

    ; Последовательность переменных
    <VariableSequence> := <Variable> (': <Variable>)*

    ; Шаблон внутри инструкции, требующей дополнительный аргумент
справа
    <InstructionTemplateWithArgument>       :=       <Operator>       |
<UnificationWithArgument>

    ; Оператор
    <Operator> := <Modifier> | <Foreach> | <Matches> | cont | one |
debug | cache

    ; Модификатор
    <Modifier> := (on | off |) <Mode>

    ; Режим преобразования
    <Mode> := ignoresp | keepinitiator | left | m | match | mvar |
multi | lazy | line | lrec | opt | ci

    ; Оператор foreach
    <Foreach> := foreach', <InstructionVariable>, <FunctionCall>

    ; Оператор matches
```

```
    <Matches> := matches', <Query>

    ; Унификация с аргументом
    <UnificationWithArgument> := <Template> '=

    ; Строка текста без спецсимволов
    <BasicString> := (<NormalCharacter> | <Quoted> | <DoubleQuoted>)+
    <NormalCharacter> := [line]. & !<SpecialCharacter>
    <SpecialCharacter> := ; Любой не буквенный или не цифровой символ Юникода; определение этого множества здесь полностью не приводится

    ; Символ, экранированный апострофом
    <Quoted> := '' .

    ; Строка в кавычках (допускает внутри себя двойную кавычку, экранированную апострофом)
    <DoubleQuoted> := '" (. & ! '" | ''")* '"
```

Синтаксис шаблона результата

Шаблоны входа предназначены исключительно для проверки входных данных на допустимость. Они не вычисляют результат. Основной же целью преобразования является построение результирующих данных. Описание этих данных задаётся в правых частях правил преобразования в виде шаблонов результата и является таким же важным, как и описание шаблонов входа. Для описания шаблонов результата используется та же нотация, что и для шаблонов входа. Но при построении результата производится уже не проверка, а формирование результирующего потока данных согласно следующей таблице.

Возвращаемые результаты

Шаблон	Результат
(пустой предикат)	(пустая цепочка)
Символ	этот символ
.	не используется в результате
^	не используется в результате
$	не используется в результате
_	не используется в результате
Конкатенация	конкатенация результатов операндов
&	не используется в результате
|	не используется в результате

!	не используется в результате
?	не используется в результате
+	не используется в результате
*	не используется в результате
#	указывает, сколько раз следует повторить результат. Допускается лишь конкретное число.
<>	не используется в результате
,	список результатов операндов
@	результат вызова функции
[]	допускаются только унификации и использования переменных
{}	этот объект
:	используется внутри объекта: задаёт значение свойства объекта

Приоритеты шаблонов

Приоритет	**Шаблон**
0	:=
1	=>, =, <=
2	,
3	:
4	|
5	&
6	\
7	конкатенация
8	!
9	символ
10	?, +, *, #
11	[]
12	@
13	-
14	{}, <>

Приоритеты шаблонов внутри инструкций

Приоритет	Шаблон
0	=
1	<, >, !=
4	+, -
5	*, /, %

Таблица соответствия шаблонов SPARD предикатам Пролога и логическим функциям JavaScript

В таблице для раскрытия сути предикатов используется нотация языка Пролог. Имена предикатов совпадают с их обозначениями в SPARD, а параметром предиката является совпадающая часть входной цепочки, обозначаемая x.

Дополнительно также даны аналоги предикатов в виде логических функций на языке JavaScript.

Шаблон	Пример	Аналог на Прологе	Аналог на JavaScript
(пустой шаблон)		$()(x).$	true
Символ	a	$a(x) :- x = 'a'.$	$x == 'a'$
	‘;	$';(x) :- x = ';'$	$x == ';'$
Любой элемент	.	$.(x) :- length(x, 1).$	$x.length == 1$
Начало строки	^	аналога нет	аналога нет
Конец строки	$	аналога нет	аналога нет
_	_	$\_(x).$	true
Комментарий	; это комментарий	% это комментарий	// это комментарий
Конкатенация	$a_1a_2 \dots a_n$	$(a_1a_2 \dots a_n)(x) :- x = x_1x_2 \dots x_n, a_1(x_1), a_2(x_2), \dots a_n(x_n).$	аналог достаточно трудоёмок
Строка	“()”	“()”$(x) :- x =$ “()”.	$x ==$ “()”

И	$A\&B$	$(A\&B)(x) :- A(x), B(x).$	$A(x)\ \&\&\ B(x)$
ИЛИ	$A\|B$	$(A\|B)(x) :- A(x);\ B(x).$	$A(x)\ ||\ B(x)$
НЕ	$!A$	$(!A)(x) :- A(x), fail.$	$!A(x)$
?	? определяется через логические операции		
+	+ определяется через логические операции		
*	* определяется через логические операции		
#	$A\#B$	$(A\#B)(x): -\big(AAA \dots A\ (n \text{ раз})\big)(x), B(n)$	аналог достаточно трудоёмок
Интервал	$A - B$	$(A - B)(x): -x \geq A, x \leq B.$	$x \geq A\ \&\&\ x \leq B$
Множество	$< A >, < A >:= P$	$< A > (x) :- Def(A, P), P(x)$ где $Def(A, P)$, если существует определение $< A >:= P$	полноценного аналога нет
Список	$A_1, A_2, \dots, A_n$	$(A_1, A_2, \dots, A_n)(x) :- x = [x_1 \| x_2 \| \dots \| x_n],$ $A_1(x_1), A_2(x_2), \dots, A_n(x_n)$	$[A_1(x[0]), A_2(x[1]), \dots,$ $A_n(x[n-1])$
Использование переменной	$[Y]$	$[Y](x) :- x = Y$	$x == Y$

Вызов функции и большинство инструкций не определяют предикаты.

Примеры правил SPARD

Ниже даны некоторые правила на языке SPARD и их описания. Эти примеры позволяют начинающим лучше понять стилистику написания программ на SPARD.

Правило	Описание
`a => b`	Замена символа a на b
`. => c`	Замена любого символа на c
`. & !  => c`	Замена любого символа кроме \r и \n на c
`[line]. => c`	Замена любого символа кроме \r и \n на c
`'« | '» | '„ | '“ => '"`	Замена типографских кавычек на обычные
`<SP>+ => " "`	Удаление группы пробелов с заменой их одним пробелом
`(^ +|) ^ОТВЕТЫ? =>  Ответ`	Замена слова «ОТВЕТ» и «ОТВЕТЫ» на «Ответ» с новой строки; удаление при этом некоторого количества идущих перед ним пустых строк
`abc => x`	Замена строки abc на x

`a | b | c => x`	Замена любого из символов a, b, c на x
`a.&.b => c`	Замена строки ab на c
`a & b => x`	Это правило никогда не применяется
`!a => b`	Замена любого символа кроме a на b.
`. & !a => b`	Замена любого символа кроме a на b
`.. & !(ab) => c`	Замена любой пары символов, не равных ab, на c
`.. & !a. & .!b => c`	Замена любой пары символов, не начинающейся на a и не оканчивающейся на b, на c
`<NotAB> := . & !a & !b` `<NotAB><NotAB> => c`	Замена любой пары символов, не содержащих a и b, на c
`[X=](. & !a & !b)[X] => c`	Замена двух одинаковых символов, не равных a или b, на c
`^a => b`	Замена символа a на b при условии, что он находится в начале текста
`^<SP>*a => b`	Более гибкий вариант предыдущего примера, учитывающий возможное наличие пробельных символов в строке
`v$ => y`	Замена символа v на y при условии, что он находится в конце текста
`[line](^.$) =>`	Удаление строк, состоящих из одного символа
`[X=]. => [X][X]`	Удвоение символа
`[X=](.+) => [X][X]`	Удвоение строки
`([X=].)+ => [X][X]`	Удвоение строки при условии, что она состоит из одинаковых символов
`[X=]. "+" [Y=]. [Z = X + Y] => [Z]`	Сложение двух символов (они должны быть цифрами)
`f := 0 => 1` `f:= [N=](.+) [M = N – 1] [R = N * @(f, M)] => [R]`	Факториал
`addPoint := [P=]. [X=](.*) [Sign=]('.| '?| '!) <SP>* $ => @(upper, [P]) [X] [Sign]` `addPoint := [P=]. [lazy][X=](.*) <SP>* $ => @(upper, [P]) [X] '.`	Функция, добавляющая точку в конец строки и превращающая первую букву строки в прописную
`[X=].* => @(cut, [X])` `cut := [T=]([X=](.*) [@(length, X) < 73] <SP> [Y=](.*)) [@(length, T) >`	Функция, разбивающая длинные строки на строки без переносов, каждая длиной не более

72] => [X] @(cut, [Y]) cut := [X=](.*) => [X]	72 символов
<A> := a<A> | a	Определение цепочки, состоящей из произвольного числа символов a (аналогично a+)
<Number> :=(1|2|3|4|5|6|7|8|9|0)+	Определение числа
<Number> := 0-9+	Альтернативное определение числа
<Number> := <d>+	Ещё одно альтернативное определение числа
<Number> := 1 | [N=](.+) [M = N - 1] [@(isnumber, M) = 1] isnumber := <Number> => 1 isnumber := _ => 0	Рекурсивное определение натурального числа (очевидно, неудачное)
a**	То же, что a*
a*+	То же, что a*
a+*	То же, что a*
a++	То же, что a+
a??	То же, что a?
ab?	a или ab
.* & !(a+)	Последовательность символов, не начинающаяся с последовательности символов a (то же, что и .*&!a и .*&!(a.*))