SPARD – язык для преобразования текстов

Автор – Хиль В.А.
По всем вопросам и предложениями просьба обращаться по следующим контактам:

Веб-сайт: http://vladimirkhil.com

E-mail: vladimir.khil at gmail dot com

Все ваши замечания будут внимательно рассмотрены.

Оглавление

Общие сведения о SPARD 4

Краткое описание синтаксиса SPARD 8

Символьные предикаты....8

Конкатенация....9

Логические операции 9

Кванторы 10

Множества 10

Функции....12

Инструкции....11

Реализация других механизмов обработки данных посредством SPARD....12

Регулярные выражения 12

Контекстно-свободные грамматики....13

Контекстно-зависимые грамматики 14

SPARD как логический язык программирования....14

Сравнение SPARD с аналогами....15

РЕФАЛ (2)....15

Perl Regexp-Grammars (3)....16

Peg и Leg (5)....18

AWK (6) 19

Chomski (7)....19

F# (8)....20

TREE-META (9) 21

COMIT (10) 22

Примеры преобразований SPARD 23

Преобразование текстов (11) 23

Извлечение данных (12) 27

Работа с естественным языком 31

Перспективы развития SPARD 34

Список использованной литературы 35

Ссылки по теме 35

Общие сведения о SPARD

В настоящий момент человечество накопило уже достаточно большой объём знаний в Интернете, представленных в виде неразмеченного (для машины) текста на естественном языке. Количество информации в сети настолько велико, что ручная обработка её не представляется возможной. Возникает естественная потребность работать с подобными текстами и извлекать из них необходимую информацию с помощью программных средств. Решение этой задачи позволит использовать данные существующих документов без необходимости их дополнительной разметки для машинного обработчика.

В качестве наиболее распространённых задач обработки текстовой информации можно назвать:

- поиск
- извлечение информации
- автореферирование
- перевод
- создание вопросов по тексту
- форматирование текста
- компиляция и метапрограммирование

Все они могут быть сведены к задаче преобразования текстов, на входе которой имеется некоторый текст или множество текстов, и на выходе должны получиться также один или несколько текстов.

Задачу преобразования решает преобразователь, имеющий вход, куда поступает исходная информация, выход, на который подаётся результат, и набор правил, управляющих преобразованием. Преобразователь абстрагируется от способа ввода в память машины исходной информации и способа предоставления результата конечному потребителю; это позволяет максимально сконцентрироваться на основной задаче преобразования входных данных.

Процесс преобразования нескольких документов представляет собой обобщение процесса преобразования одного документа, поэтому основной интерес представляет простой преобразователь, преобразующий единичный текст в единичный текст.

Для описания правил преобразования и был придуман язык **SPARD** – **String processing Pattern matching Rule-based Data-driven language** (язык для описаний преобразований данных). В процессе разработки языка к нему были выдвинуты следующие требования:

1) объём входного и выходного документы неограниченны;

2) правила преобразования должны быть понятными для человека настолько, насколько это возможно;

3) при этом программы на языке должны быть короткими;

4) каждое правило должно описывать простое преобразование, а сложное преобразование должно строиться из большого набора простых;

5) сами языковые конструкции должны быть простыми (требование, банально упрощающее реализацию языка).

В итоге необходимо было получить декларативный язык, позволяющий описывать большое количество достаточно простых правил. Декларативность как раз и обеспечивает удобство восприятия текстов на SPARD при одновременном усложнении работы преобразователя.

Декларативный язык описания преобразований уже существует в настоящее время, это язык XSLT (1). Он используется для описания преобразований XML-документов в произвольные текстовые файлы. Поэтому основной принцип работы XSLT-процессора был перенесён в SPARD.

В основе SPARD-преобразования лежит алгоритм сравнения с образцом: преобразователь содержит набор шаблонов, с которым сравнивается входной документ в текущей позиции; на основании совпавшего шаблона выбирается результат преобразования.

Типовое правило в программе на SPARD имеет вид[1]

```
Template => Result
```

[1] ср. с XSLT:
<xsl:template match="Template">
Result
</xsl:template>

где `Template` – шаблон, с которым сравнивается входной текст (или, иными словами, в текущей позиции входных данных проверяется соответствующий шаблону предикат), `Result` – результат преобразования.

Запись предельно понятна (видно, что во что преобразуется) и ориентирована на неподготовленного читателя. `Template` в данном контексте можно понимать в трёх смыслах:

1. Это шаблон, с которым сравниваются входные данные.
2. Это предикат, который должен выполниться в текущей позиции входных данных.
3. Это множество, задающее цепочки символов, которые принимаются данным правилом.

Все три толкования по сути дела означают одно и то же: производится тест входных данных в текущей позиции на предмет выполнения некоторых условий. В случае выполнения этих условий происходит применение правила – на выход выдаётся `Result`.

SPARD-преобразователь получает на входе последовательный текст и может перемещаться только вперёд. Его задача – осуществить построение результирующего текста на основе заложенных в него правил обработки.

Таким образом, SPARD-преобразователь содержит считывающую головку, которая в начале работы находится перед первым символом текста. Этот символ называется текущим символом. Головка может смещаться вперёд по тексту без возможности возвратов.

Можно сформулировать общий алгоритм работы SPARD преобразователя, использующего набор правил указанного вида `Template => Result`.

1. На вход преобразователю подаётся текст, текущим символом является первый символ текста.
2. Преобразователь последовательно проверяет предикаты всех имеющихся у него правил в порядке их следования; как только один из предикатов выполняется, преобразователь выдаёт в выходной поток соответствующий результат и сдвигает позицию во входном тексте на число совпавших символов. Происходит возврат к шагу 1.

3. Преобразователь останавливается, если ни один предикат правила не был выполнен в текущей позиции. Если при этом был прочитан весь входной текст, преобразование считается выполненным успешно. В противном случае фиксируется ошибка преобразования.

Рассмотрим пример работы преобразователя.

Преобразование задано правилами:

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

На вход преобразователю подаётся текст “abba”.

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 образцами (проверять выполнение предикатов). В данном примере проверка предикатов заключаются в проверке строгого совпадения входа с образцом. Таким образом, будет истинен предикат в правиле (1) (вход содержит цепочку “ab”), и на выход будет выдан 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 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».

Таким образом, преобразователь совершает общее преобразование текста, используя локальные преобразования, заданные отдельными правилами. Такой подход позволяет достаточно простым образом описывать сложные преобразования данных.

SPARD – не язык запросов к тексту наподобие SQL и не язык, автоматизирующий работу с естественноязыковыми текстами. SPARD предоставляет низкоуровневый механизм работы с текстами, на основе которого можно разрабатывать упомянутые выше механизмы. Он решает вполне мирские задачи, о которых речь пойдёт ниже.

Краткое описание синтаксиса SPARD

Вся мощь языка SPARD заключается в большом количестве используемых шаблонов, позволяющих производить достаточно сложные тесты входных данных. Полное описание возможностей языка дано в его спецификации, здесь же указаны лишь основные его возможности.

Каждый шаблон SPARD задаёт соответствующий предикат, проверяющий входные данные в текущей позиции. Шаблоны можно комбинировать между собой, создавая более сложные предикаты.

Символьные шаблоны

Базовым шаблоном для SPARD является символ. Такой шаблон требует наличия во входном тексте этого конкретного символа. К примеру, предикат шаблона `a` будет истинен только в том случае, если текущим символом входного текста является `a`.

Чтобы указать, что в текущей позиции входного документа может находиться любой символ (кроме символов переноса строки), нужно использовать шаблон `.`. Следующее правило SPARD преобразует произвольный символ в символ `x`:

```
. => x
```

Допустим, мы хотим построить преобразователь, выполняющий быструю замену строк во входном тексте. На языке SPARD он записывается очень просто:

```
“т.е.” => “то есть”
“и т.д.” => “и так далее”
“и т.п.” => “и тому подобное”
```

Все преобразования будут выполнены за один проход преобразователя по входному тексту. Кавычки позволяют использовать в шаблоне символы без их специального значения (точка обозначает точку во входном тексте, а не произвольный символ). Можно экранировать действие одиночного спецсимвола, поставив перед ним апостроф `‘`. Лучше всего экранировать все не буквенно-цифровые символы, так как они могут иметь специальный смысл либо в текущей, либо в последующей версиях языка.

Конкатенация

Конкатенация – основная операция при работе с текстовыми данными. Она позволяет «склеивать» шаблоны друг с другом, образуя составные шаблоны. В SPARD конкатенация не обозначается никаким специальным символом; для её осуществления достаточно записать шаблоны друг за другом.

В предыдущем примере конкатенация уже использовалась: там были последовательно записаны шаблоны-символы, в результате чего получилась шаблон-строка. Предикат, определяемый данным шаблоном, выполняется только в том случае, если входной текст в текущей позиции начинается с этой подстроки.

Конкатенация – удобный механизм проверки предикатов, налагающих условия на несколько символов входного текста. Наибольшая эффективность конкатенации проявляется в случаях, когда длина цепочки, которая должна удовлетворять предикатам-операндам, заранее неизвестна. Например, можно рассмотреть предикат `(a|bc)(b|cd)(c|de)`, представляющий собой конкатенацию трёх шаблонов, каждый из которых может совпасть либо с одним, либо с двумя символами. При этом предикат будет выполняться на цепочках `abc`, `acdc`, `bccdde` и др. В реальных задачах часто встречаются подобные примеры, когда необходимо разделить входные данные на подцепочки с условием, чтобы на каждой подцепочке выполнялся отдельный предикат. Программировать такую функциональность каждый раз достаточно накладно, а SPARD предоставляет встроенный механизм такого разбиения входного текста.

Логические операции

Логические операции частично уже использовались выше. Они позволяют комбинировать налагаемые на входной текст условия так же, как это принято в большинстве языков программирования.

Шаблон `|` обозначает операцию ИЛИ, шаблон `&` - операцию И, шаблон `!` – операцию НЕ. Отрицание в SPARD следует использовать с той же осторожностью, как и в языке Пролог.

Вот пример шаблона, обозначающего десятичные цифры:

```
0|1|2|3|4|5|6|7|8|9
```

Вот пример шаблона, допускающего любые пары символов, не начиняющиеся на a и не заканчивающиеся на b:

```
(.&!a)(.&!b)
```

Шаблон, допускающий любые пары символов кроме ab:

```
..&!(ab)
```

Конкатенация имеет больший приоритет по сравнению с операцией &, поэтому в данном случае операнды не заключены в скобки. Операция ! имеет более высокий приоритет, нежели конкатенация, и поэтому в её случае скобки требуются.

Квантификаторы

Квантификаторы позволяют указать необходимость присутствия того или иного шаблона в цепочке определённое количество раз: один раз, произвольное число раз или ни одного.

В SPARD используются четыре общеизвестные квантификатора ? (ноль или один раз), + (один и более раз), * (ноль и более раз) и # (число раз, заданное шаблоном). Использование квантификаторов делает шаблоны более выразительными, хотя любой из квантификаторов можно определить через обычные логические операции.

Конструкция [line](.+) позволяет описать произвольную строку, не содержащую символов переноса строки[2], а шаблон .+ – произвольный текст вообще.

Теперь мы можем определить шаблон, задающий натуральное число:

```
(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
```

Множества

Очень часто в разных местах правил SPARD приходится использовать один и тот же шаблон, поэтому удобно дать ему имя и ссылаться на определение по имени. Именованный шаблон называется множеством. Вообще говоря, любой шаблон обозначает множество цепочек, удовлетворяющих ему, но в контексте SPARD имеет смысл говорить о множествах как об именованных шаблонах.

Определение множества записывается в отдельной строке файла преобразования SPARD. Определим, к примеру, цифру и натуральное число:

---

[2] line – модификатор, применение которого меняет логику работы шаблона . (точки). В этом режиме . не совпадает с символами переноса строки. Модификаторы рассмотрены подробно далее.

```
<Digit> := 0|1|2|3|4|5|6|7|8|9
<Number> := (<Digit>&!0)<Digit>
```

Благодаря использованию множеств, запись стала короче и интуитивно понятнее.

Множества могут иметь параметры. Они могут быть шаблонами или переменными, записываемыми в квадратных скобках. Опишем, например, множество, состоящее из цепочек, содержащих два одинаковых символа.

```
<Pair, [X]> := [X][X]
```

Теперь при описании шаблонов можно использовать, к примеру, множество `<Pair, a>`, обозначающее два символа `a` подряд.

Инструкции

С помощью инструкций можно различными способами модифицировать ход преобразования текста. Главная функция инструкций – это возможность использования переменных.

Очень часто бывает так, что совпавшую подцепочку из входного текста необходимо использовать для каких-то вычислений или передать в качестве результата преобразования. Для этих целей и служат переменные. Они позволяют занести в именованную область памяти совпавшую подцепочку и затем ссылаться на неё. Вот пример использования переменных, удваивающий входной текст:

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

Знак равенства обозначает унификацию. Когда справа от равенства отсутствует значение, переменная унифицируется с совпавшей подцепочкой, удовлетворяющей шаблону справа от инструкции. Символ _ представляет собой предикат, совпадающий со всем оставшимся текстом. Простое указание имени переменной в результате преобразования выдаёт значение переменной в выходной поток.

Переменную можно использовать и в шаблоне для того, чтобы потребовать соответствия входной цепочки значению этой переменной. Например, так можно потребовать наличия во входном тексте двух одинаковых символов:

```
[X=].[X]
```

Унификацию можно использовать для задания переменной произвольных значений и для унификации переменных между собой:

```
[A=B][A=4]_ => [B]
```

Функции

Часто требуется повторно использовать не только шаблон-предикат, но и результат преобразования. Для этой цели в SPARD существуют функции, которые используются как и в остальных языках программирования: у функции должно быть определение и должен быть вызов с передачей соответствующих параметров.

Функция представляет собой внутренний автономный SPARD-преобразователь, работающий по своему набору правил. Использование такого преобразователя позволяет отделить определённую логику преобразования, если она актуальна только на определённом участке входных данных. Ну и функции можно просто использовать для вычисления какого-либо вспомогательного результата.

Вычислим факториал на SPARD.

```
fact := 0             => 1
fact := [N=]<Number> => [N1 = N – 1][N * @(fact, N1)]
```

Существуют и другие многочисленные варианты использования инструкций и другие конструкции SPARD, но полное описание языка дано в его спецификации. Здесь же изложены базовые механизмы, дающие общее представление о структуре языка, его возможностях и стиле программирования на нём. Далее будут приведены реальные примеры задач, для решения которых использовался SPARD.

Реализация других механизмов обработки данных посредством SPARD

С помощью SPARD можно описывать конструкции, уже известные в математике и программировании. SPARD обобщает их и предоставляет дополнительные возможности, упрощающие жизнь разработчику.

Регулярные выражения

Любой современный язык программирования поддерживает регулярные выражения (РВ). Как правило, с помощью РВ удаётся проверить соответствие строки некоторому шаблону, выполнить замену в строке и извлечь из строки какие-либо данные.

Преимуществом использования регулярных выражений является их встроенность в язык программирования. Недостаток – невозможность описания сложных преобразований, зависящих от входных данных (`a` заменить на `b`, а `c` – на `d`).

SPARD позволяет описывать регулярные множества с помощью шаблонов-предикатов, полностью аналогичных регулярным выражениям. Достаточно проверить, что синтаксис SPARD удовлетворяет определению регулярных выражений Ахо – Ульмана (1 стр. 124).

1. Пустое множество не обозначается никаким шаблоном.
2. Множество из пустого элемента описывается с помощью пустого шаблона.
3. Множество, содержащее символ a, обозначается как a.
4. Объединение, конкатенация и рекурсивное замыкание регулярных множеств обозначаются соответственно как A|B, AB и A*.

Таким образом, шаблоны SPARD обобщают понятие регулярного множества, позволяя, помимо всего перечисленного, использовать гораздо более сложные конструкции. В этом случае можно описывать нерегулярные множества. Задачи проверки соответствия шаблону, извлечения данных и их преобразования также полностью решаются средствами SPARD.

Контекстно-свободные грамматики

Контекстно-свободные грамматики (КСГ) – более мощный механизм описания множеств. Множество задаётся совокупностью правил вывода цепочек из начального нетерминального символа. Поддержка КСГ слабо представлена в современных языках программирования, но легко реализуется посредством множеств языка SPARD.

Терминальные символы при этом соответствуют простым предикатам SPARD, состоящим из этих символов. Нетерминалы соответствуют множествам SPARD.

Например, правило КСГ

```
S -> aAS|b
```

в SPARD будет записано как

```
<S> := a<A><S>|b
```

Программисту не нужно заботиться о реализации механизма распознавания входных цепочек по заданной грамматике, преобразователь SPARD это сделает за него. Существуют программные средства, позволяющие генерировать проверяющий код по

заданной входной грамматике, но SPARD предоставляет большие возможности преобразования, не укладывающиеся в рамки КСГ.

Контекстно-зависимые грамматики

SPARD позволяет описывать и контекстно-зависимые грамматики (КЗГ). Они используются в тех случаях, когда возможностей КСГ не хватает. Правила КЗГ позволяют записывать в левой части правил дополнительные символы, влияющие на работу преобразователя. SPARD позволяет записывать контекстные символы только после преобразуемого нетерминала, но это не ограничивает общность преобразователя.

Прямого соответствия между правилами вывода КЗГ и правилами SPARD нет, для получения правил SPARD нужно выполнить дополнительные преобразования. Допустим, необходимо в SPARD определить контекстно-зависимое множество цепочек $\{a^n b^n c^n | \ n \geq 1\}$. В SPARD можно указать необходимое количество раз выполнения того или иного предиката:

```
a#([N=]<Int>) b#[N] c#[N]$
```

SPARD как логический язык программирования

Предикаты и множества SPARD можно использовать так же, как и предикаты языка Пролог, т.е. строить высказывания. Именованным предикатам Пролога соответствуют множества SPARD, а формальные и фактические параметры множеств связываются между собой посредством унификации.

Возьмём пример предиката на Прологе, выполняющего проверку, подано ли на вход число:

```
Digit(X) := X >= 0, X =< 9.
Number(X) :- Digit(X).
Number([X|T]) :- Digit(X), Number(T).
```

На SPARD подобный предикат можно записать так:

```
<Digit> := 0-9
<Number> := <Digit>(<Number>|)
```

Вот пример предиката Пролога, разворачивающего список с использованием хвостовой рекурсии:

```
R([], L, L).
```

```
R([H|T], L1, L2) :- R(T, [H|L1], L2).
```

В SPARD мы можем описать не множество, а функцию (что более логично, так как производится не проверка, а преобразование данных), обращающую входную цепочку символов с помощью рекурсии:

```
R := [H:T] => @(R, [T])[H]
R := [H]   => [H]
```

Сравнение SPARD с аналогами

Разумеется, SPARD и XSLT – не единственные существующие на сегодняшний день средства по преобразованию текстов. Можно отметить альтернативные технологии, сравнить их с SPARD и определить их преимущества и недостатки.

РЕФАЛ (2)

РЕФАЛ – язык символьных преобразований, основанный на алгорифмах Маркова. Он использует функции, каждая из которых содержит набор правил. Правило – это пара из шаблона и результата. В момент вычисления функции входное текстовое выражение сравнивается с шаблонами правил в порядке их определения. В случае совпадения выражения с каким-либо из шаблонов на выход функции подаётся результат, соответствующий данному шаблону. В случае невозможности обнаружить подходящий для входа шаблон выполнение функции признаётся неуспешным.

Таким образом, SPARD-функция очень похожа на РЕФАЛ-функцию. Важное отличие между ними заключается в том, что РЕФАЛ-функция подбирает образец подо всё входное выражение целиком, а SPARD-функция ищет некоторую подстроку, с которой начинается входной текст в текущей позиции и которая удовлетворяет одному из шаблонов. Впрочем, эти два подхода эквивалентны: достаточно добавить в SPARD концевой шаблон _, «съедающий» весь остаток входного текста, и правила преобразования будут идентичны (но только при нахождении считывающей головки SPARD в начале текста).

Шаблон передаёт данные в результат через переменные, которые в этом языке типизированы (единичные символы, термы и произвольные выражения). Особенностью РЕФАЛ-преобразования является то, что результат очень часто представляет собой не просто последовательность символов и переменных, но и содержит ссылки на другие функции, которые должны быть вычислены впоследствии.

Основное отличие между SPARD и РЕФАЛом заключается в порядке выполнения общего преобразования текста. Если SPARD-преобразователь представляет собой глобальную функцию, единожды читающую входной текст и выдающую результат по мере его формирования, то РЕФАЛ-машина действует по принципам алгорифма Маркова: она неограниченное число раз изменяет входное выражение до тех пор, пока в нём будут присутствовать ссылки на другие функции. Эти функции вызываются, вместо них поставляются результаты их вызова (возможно, содержащие вызовы других функций). В результате выполнения этого циклического процесса на выходе получается искомый результат.

В SPARD подобное многопроходное преобразование достигается, как и во всех функциональных языках, с помощью рекурсивных функций.

Возьмём пример вычисления палиндрома на РЕФАЛе из Википедии и напишем аналогичную функцию на SPARD:

```
Palindrom {
     s.1 e.2 s.1 = <Palindrom e.2> ;
     s.1 = True ;
     = True;
     e.1 = False ;
 }
```

SPARD:

```
palindrom := [X=].[Y=]<String>[X]$ => @(palindrom, [Y])
palindrom := .$ => True
palindrom := ^$ => True
palindrom := _ => False
```

В целом, идеологически языки очень близки друг другу, но SPARD предоставляет больше возможностей (например, богатый язык регулярных выражений) для описания шаблонов входных данных.

Perl Regexp-Grammars (3)

Язык Perl имеет богатые возможности для работы с текстом за счёт использования регулярных выражений. Библиотека Regexp-Grammars расширяет встроенные возможности языка, делая его по-настоящему мощным средством для работы с текстами.

Regexp-Grammars позволяет описывать не только регулярные выражения, но и иерархически организованные структуры с использованием тех же регулярных

выражений. По сути дела, любому регулярному выражению становится возможно присвоить имя и ссылаться на него в другом регулярном выражении. В результате получается синтаксис, эквивалентный синтаксису конечных грамматик. Использование конечных грамматик значительно расширяет возможности программиста.

Парсер Rerl использует заданную иерархию правил и пытается разобрать входной текст в соответствии с заданными правилами. На выходе получается иерархически организованная структура данных (дерево), вершины которой соответствуют совпавшим со входом регулярным выражениям.

Regexp-Grammars определяет также переменные, с помощью которых возможно потребовать наличия во входном тексте например двух одинаковых подряд идущих символов.

Структурой-аналогом для правил Perl является множество SPARD. Оно также представляет собой именованное предикат-выражение (и также допускает наличие параметров), на которое можно ссылаться в определениях других множеств. Таким образом, базовое правило преобразования на Regexp-Grammars может быть легко перенесено на SPARD.

Рассмотрим пример разбора простейшего предложения на Regexp-Grammars:

```
<rule: sentence>
        <noun> <verb> <object>

<rule: noun>
      \s+

<rule:verb>
      \s+

<rule:object>
      \s+
```

В SPARD аналогичное определение будет записано так:

```
<Sentence> := <Noun><SP>+<Verb><SP>+<Object>
<Noun> := <Letter>+
<Verb> := <Letter>+
<Object> := <Letter>+
```

Дополнительно придётся определить только множество `<Letter>`.

Основным преимуществом SPARD перед Perl является большая понятность правил преобразования.

Вот так записывается замена подстроки `abc` на `def` во всей входной строке на языке Perl:

```
s/abc/def/g
```

А вот как это же преобразование выглядит на SPARD:

```
abc => def
```

Запись на SPARD более наглядна.

В отличие от Perl'а SPARD не является полноценным языком программирования, в нём нет операций ввода-вывода, взаимодействия с файловой системой и пр. Он ориентирован исключительно на задачу преобразования данных и за счёт этого обеспечивает высокую эффективность в решении данной задачи. Отсутствие привязки к внешним условиям функционирования SPARD-преобразователя делает его полностью кроссплатформенным.

Peg и Leg (5)

Peg и leg – два схожих генератора парсеров. На вход они принимают описание грамматики в виде набора правил вывода (в собственном формате), на выходе получается функция на языке C++, разбирающая текст. Сгенерированная функция может быть затем использована в программах для разбора текстов и извлечения данных.

Функцию можно вызывать неоднократно, и она при каждом вызове будет анализировать ещё не прочитанную часть входного текста до тех пор, пока текст не будет прочитан целиком или пока разбор не завершится с ошибкой.

Правила грамматики позволяют ассоциировать с собой код на языке C++, выполняемый в случае совпадения шаблона правила с входным текстом. Этот механизм расширяет возможности парсера до полноценного преобразователя.

SPARD позволяет описывать аналогичные грамматики с использованием множеств, но не позволяет генерировать исходный код для парсеров на других языках программирования. SPARD разработан с ориентацией исключительно на преобразования, не принимая в расчёт условия, в которых этих преобразования выполняются. Любая дополнительная функциональность может быть внедрена в конкретных реализациях

SPARD-преобразователей. В том числе возможно реализовать и генерацию кода, но в спецификацию SPARD это не входит.

AWK (6)

AWK – хороший пример языка программирования, позволяющего организовывать вычисления, управляемые данными. Как и в синтаксисе peg, программа на AWK выполняет очередное действие в соответствии с тем, какому из шаблонов удовлетворяют входные данные. Для AWK такие данные – это заранее структурированный текст (увы, серьёзный недостаток).

Каждое правило AWK записывается как `шаблон {действие}` и располагается на отдельной строке. Например:

```
/Hello/ { print “Hello, %username%” }
```

Связка «шаблон – действие» предоставляет программисту гораздо большие возможности, нежели связка «шаблон – результат», ведь действие может заключаться не только в выдаче результирующих данных, но и, например, в выдаче пользователю сообщений, в работе с файловой системой и сетью. Однако данное расширение смешивает процесс преобразования с процессом взаимодействия преобразователя с окружающей средой. Функция преобразования перестаёт быть функцией в математическом смысле, а язык перестаёт быть функциональным. В SPARD было принято решение строго следовать концепции функционального языка и чистых функций, а также сосредоточиться исключительно на самом процессе преобразования. Отсутствие императивной логики и понятия действия в языке делает его полностью декларативным, что обеспечивает возможность оптимизации всех этапов преобразования.

Таким образом, если программа на AWK может быть скомпилирована в полноценный исполняемый файл, решающий поставленную задачу, SPARD-программа представляет собой лишь модуль, который включается в другие приложения и выполняет исключительно задачу преобразования. Данное отличие языков мы будем трактовать как недостаток AWK: ведь он привязан к конкретному языку, на котором пишутся команды взаимодействия с окружением, и к конкретным условиям выполнения программы (наличие консоли для ввода и вывода данных).

Chomski (7)

Chomski – программная утилита, названная в честь Ноама Хомского.

Принципы работы этой утилиты аналогичны принципам peg и AWK, то есть в основе преобразований лежит правило в форме «шаблон – действие». Утилита используется в вызовах из командной строки, она читает стандартный поток ввода и пишет в стандартный поток вывода.

Функциональные возможности данной утилиты невелики, но она вполне позволяет выполнять простейшие действия по замене символов текста и извлечения из него данных.

Следующее выражение используется в Chomski для выделения из входного текста только тех фрагментов, которые записаны в скобках:

```
'/(/ { until ")"; print; } clear;'
```

Та же самая функциональность доступна и в SPARD:

```
[X=]("("<String>")") => [X]
. =>
```

На самом деле любая задача, решаемая с помощью Chomski, может быть решена на SPARD, причём запись правил на SPARD не будет особенно длиннее записи правил для Chomski.

F# (8)

F# в данной статье выбран как один из представителей языков функционального программирования. Не касаясь практически всех возможностей языка, обратим внимание лишь на реализацию механизма сравнения с шаблонами в нём.

Любой функциональный язык поддерживает механизм сравнения с шаблонами (образцами). Этот механизм реализуется посредством оператора или функции, принимающей на вход некоторую переменную и умеющую сравнивать переменную с выражениями (в порядке их определения). После первого успешного сравнения функция возвращает результат, ассоциированный с заданным шаблоном. Все функции SPARD работают аналогичным образом.

В F# механизм сравнения с образцом реализован оператором match. В общем виде его использование выглядит так:

```
match test-expression with
    | pattern1 [ when condition ] -> result-expression1
    | pattern2 [ when condition ] -> result-expression2
    | ...
```

Для каждого шаблона допускается дополнительное указание условий совпадения, что позволяет проверять не только наличие в тестовом выражении каких-то необходимых элементов, но и проводить дополнительные проверки (сравнивать символы между собой, проводить вычисления и пр.). Это делает механизм шаблонов более мощным.

Язык шаблонов F# весьма богат и позволяет описывать большой набор тестов входных данных. Подробнее об этом можно почитать в спецификации языка.

SPARD имеет полностью аналогичную структуру шаблонов с образцами, результатами и инструкциями, представляющими собой как раз наложение дополнительных условий на тестовое выражение. В SPARD инструкции могут находиться где угодно, а не обязательно в конце шаблона. Поэтому и проверяться они могут до завершения полного сравнения выражения с образцом.

```
test := pattern1 [condition1] => result1
test := pattern2 [condition2] => result2
…
```

Напишем на F# проверку того, входит ли значение переменной в диапазон от 0 до 100:

```
match val with
var1 when var1 >= 0 && var1 <= 100 -> printf ‘Good’
_ -> printf ‘Bad’
```

А теперь напишем аналогичный тест на SPARD:

```
[Var1=]_[Var1 > -1][Var1 < 101]$ => “Good”
_ => “Bad”
```

или, что ещё короче:

```
0-100$ => “Good”
_ => “Bad”
```

TREE-META (9)

TREE-META – ещё один компилятор компиляторов. На вход программе подаётся описание входной контекстно-свободной грамматики, а на выходе получается компилятор языка, описанного с помощью данной грамматики.

Поскольку SPARD позволяет описывать контекстно-свободные грамматики и атрибутные грамматики с помощью множеств, правила SPARD покрывают правила TREE-

META и позволяют использовать SPARD-преобразователь в качестве основного компонента компилятора.

COMIT (10)

Язык, разработанный специально для его использования лингвистами. Синтаксис языка сделан таким, чтобы лингвист мог без помощи программиста самостоятельно заносить в систему правила, а совокупность заданных правил обеспечивала бы механизм машинного перевода.

Синтаксис языка, тем не менее, достаточно сложен для первичного ознакомления. Вся программа состоит из набора правил, описывающий всё тот же механизм сравнения с образцом и переписывания термов. Программа на COMIT последовательно перебирает занесённые в неё правила и сравнивает их левые части с входным текстом. В случае успеха программа использует информацию из правой части правила, чтобы совершить преобразование над левыми частями правила и осуществить тем самым процесс машинного перевода с одного языка на другой. После успешного выполнения правила программа не начинает перебирать правила сначала, а перемещается к заданному правилу, записанному после результата преобразования. Каждое правило COMIT имеет имя, и поэтому появляется возможность переходить между ними.

Язык шаблонов позволяет производить достаточно разнообразные тесты входных данных, но по сложности описания он заметно уступает языку шаблонов SPARD. Рассмотрим пример преобразований на двух языках:

Вот пример программы COMIT, заменяющий в тексте все символы A на B, а B – на A:

1	A = QR	1
2	B = A	2
3	QR = B	3

Хитрость преобразования в данном примере заключается в том, что сочетания QR не может встретиться в тексте на английском языке, поэтому оно используется в качестве временного заменителя для символов A. Все правила преобразования ссылаются сами на себя, поэтому они выполняются строго по очереди до тех пор, пока каждое из них может быть применимо.

Данный пример сразу показывает недостатки языка: используется искусственный заменитель, который неотличим невооружённым глазом от символов обычного текста;

отсутствуют переменные, решающие подобную задачу. Дополнительные имена правил и ссылки на них ухудшают восприятие правил преобразования и последовательности вычисления.

Ну и главный недостаток языка тот же, что и РЕФАЛа – он изменяет входные данные. Можно говорить о том, что после каждого применения правил создаётся копия входных данных, содержащая изменения, но в этом случае имеет место недопустимая производительность преобразователя.

SPARD не изменяет вход и пробегает его только один раз, и правила, решающие поставленную задачу, записываются на нём максимально компактным образом:

```
A => B
B => A
```

Примеры преобразований SPARD

В этом разделе рассмотрены три реальных случая использования SPARD-преобразователя для решения сложных задач. В примерах используется немного устаревший синтаксис языка, но общая суть преобразований понятна.

Преобразование текстов (11)

В декабре 2009 года у меня возникла реальная задача для использования SPARD. Меня попросили написать программу, форматирующую файлы с вопросами для игры «Что? Где? Когда?» (ЧГК). Вопросы составлялись различными авторами для турниров, и каждый автор оформлял вопросы по-своему. Были общие принципы форматирования, но они не требовали жёсткой структуры входных данных. Для добавления же этих файлов в Базу вопросов требовалось привести их к единому формату, который мог бы быть воспринят программой.

От программы требовалось отформатировать входной текст в соответствии с набором определённых правил. Большая часть правил включала выполнение локальных преобразования, затрагивающих лишь небольшой участок текста (поставить точку в конце строки, например). Для описания таких преобразований SPARD подходил очень хорошо. В связи с этим он и был выбран в качестве основного языка в программе.

Вот пример одного из вопросов входного файла:

Вопрос 1:
В стихотворении Андрея КОрфа покрывающие лесные отрОги подснежники именуются ИМИ весны. Назовите ИХ двумя словами, учитывая, что Питер Пен с НИМИ так и не расстался.
Ответ: молочные зубы
Комментарий: у вечно юного Питера Пена были молочные зубы.
И лету навстречу росли по отрогам лесным Подснежники марта - молочные зубы весны.
Источники: 1.http://www.relga.ru/Environ/WebObjects/tgu-www.woa/wa/Main?textid=2159&level1=main&level2=articles
2. http://piterp.ru/index.php?page=ostrov
Автор: Алексей Гноевых (Н. Новгород)

От программы требовалось вынести заголовки «Вопрос», «Ответ», «Автор» и др. в отдельные строки. Нужно также добавить, где потребуется, пустые строки, а в других местах, наоборот, – пустые строки удалить. Источники должны быть пронумерованы и все (кроме гиперссылок) заканчиваться точкой. Ну и были дополнительные моменты.

Задача требовала выполнения разнородных преобразований, затрагивающих общие участки текста. Выполнение подобных преобразований за один проход текста было затруднительным, поэтому было принято решения форматировать текст шестью последовательными SPARD-преобразователями, каждый из которых выполнял достаточно простую задачу. Первый преобразователь читал текст и возвращал результат преобразования в качестве входа для второго преобразователя, второй – преобразовывал вход и подавал результат на вход третьему и т.д. Шестой преобразователь выдавал результирующий отформатированный текст.

Данный подход также позволил уменьшить набор правил, требующихся для выполнения каждого преобразования, сделал их более понятными и позволил отключать преобразователи, если какое-либо из правил форматирования в текущей ситуации не требовалось.

Программа предоставляет возможность загружать входной файл и выгружать выходной, сохранять и загружать правила преобразований и редактировать их между выполнением преобразований.

Ниже приведены правила для всех шести преобразователей. Это первое приложение, использующее SPARD, поэтому примеры правил не являются образцом для написания. Они просто иллюстрируют подходы к решению различных типовых задач.

1. Простейшие исправления (стираются лишние пробелы, символы переноса строки и т.д.).

```
<SP>*<BR><SP><SP><SP>+ => <BR>"   "
<BR>*!. =>
<SP>*<BR><SP>* => <BR>
<SP>+ => " "
'«|'»|'„|'“ => '"
'–|'—|" - " => '—
```

2. Основной модуль, выделяющий смысловые блоки в тексте (поля вопроса, ответа, источников, авторов, комментариев). Важно отметить, что названия полей могут быть весьма экзотическими. В нужных местах также проставляются точки.

```
<Number> := (0|1|2|3|4|5|6|7|8|9)+
(^<BR>+|)("Тур      "[N=]<Number>|[N=]<Number>"      тур")      =>
<BR>"Тур:"<BR>[N]" тур"
(^<BR>+|)ВОПРОС => <BR>Вопрос
(^<BR>+|)Вопрос<SP>[N=]<Number>(':|'.)?<SP>[X=].*  =>  <BR>Вопрос'
[N]':<BR>[X]
(^<BR>+|)Вопрос<SP>[N=]<Number>(':|'.)? => <BR>Вопрос' [N]':
(^<BR>+|)Вопрос => <BR>Вопрос
<BR><BR>+[N=]<Number>'.' [X=].* => <BR><BR>Вопрос' [N]':<BR>[X]
<BR><BR>+[N=]<Number>'.? => <BR><BR>Вопрос' [N]':
(^<BR>+|)^ОТВЕТЫ? => <BR>Ответ
(^<BR>+|)^<SP>*Ответы?('.|':)<SP>*[X=].+                          =>
<BR>"Ответ:"<BR>@(addPoint, [X])
(^<SP>*<BR>+|)^Ответы?('.|':)<SP>*[X=].+                          =>
<BR>"Ответ:"<BR>@(addPoint, [X])
(^<BR>+|)^КОММЕНТАРИ(И|Й) => <BR>Комментарий:
(^<BR>+|)^Комм'.?':?<SP>[X=].*                                    =>
<BR>"Комментарий:"<BR>@(addPoint, [X])
(^<BR>+|)^Комментари(и|й)('.|':)<SP>[X=].*                        =>
<BR>"Комментарий:"<BR>@(addPoint, [X])
<SP>Комментари(и|й)('.|':)<SP>[X=].*                              =>
<BR>"Комментарий:"<BR>@(addPoint, [X])
(^<BR>+|)^Зач(ё|е)т':<SP>[X=].*   =>   <BR>"Зачет:"<BR>@(addPoint,
[X])
(^<BR>+|)^ИСТОЧНИКИ? => <BR>Источник
<SP>Источники?('.|':)<SP> => <BR>"Источник:"<BR>
(^<BR>+|)^Ист'.?':?<SP>  => <BR>"Источник:"<BR>
(^<BR>+|)^Источники?('.|':)(<SP>*<BR>?) => <BR>"Источник:"<BR>
(^<BR>+|)^АВТОРЫ? => <BR>Автор
(^<BR>+|)^Авторы?('.|':)<SP>*<BR>?[X=].*'.? => <BR>"Автор:"<BR>[X]
(^<BR>+|)^Авторы?<SP>'-<SP> => <BR>"Автор:"<BR>
(^<BR>+|)^Авторы? вопроса':<SP>[X=].*'. => <BR>"Автор:"<BR>[X]
(^<BR>+|)^Авторы? вопроса':<SP>[X=].*=> <BR>"Автор:"<BR>[X]
<SP>Авторы?('.|':)<SP>[X=].*'. => <BR>"Автор:"<BR>[X]
<SP>Авторы?('.|':)<SP>[X=].* => <BR>"Автор:"<BR>[X]
```

```
    ^<SP>*$<BR>(^&(!Вопрос&!ВОПРОС&!Ответ&!<Number>'.)) => "   "
    ^1'.([P=].&!<SP>) => 1'.' [P]
    addPoint := [X=](.*"http://".*|.*www.*)('.|',|';)$ => [X]
    addPoint := [X=](.*"http://".*|.*www.*) => [X]
    addPoint   :=   [P=].[X=].*[Sign=]('.|'?|'!)<SP>*$   =>   @(upper,
[P])[X][Sign]
    addPoint := [P=].[on, lazy][X=].*<SP>*$ => @(upper, [P])[X]'.
    work := (<SP>|<BR>)*[A=]([N=](1|2|3|4|5|6)'.<SP>*(.|<BR>)*)[N1 = N
+ 1]<BR>[X=]([N1]'.<SP>*(.|<BR>)*) => <BR>"   "@(addPoint, [A])@(work,
[X])
    work := (<SP>|<BR>)*[A=]([N=](2|3|4|5|6)'.<SP>*(.|<BR>)*) => <BR>"
"@(addPoint, [A])
    work := [X=](.|<BR>)* => <BR>@(addPoint, [X])
```

3. Основная задача следующего модуля – обработка источников. Каждый источник получает порядковый номер (существующая нумерация стирается) и завершает точкой (если не представляет собой гиперссылку).

```
    "Ответ:"<BR>[X=]((.|<BR>)&!(<BR>"Зачет:")&!(<BR>"Комментарий:")&!(
<BR>"Источник:"))* => "Ответ:"@(work, [X])<BR>
    "Источник:"<BR>[X=]((.|<BR>)&!(<BR>"Автор:"))*                  =>
"Источник:"@(work3, [X])
    addPoint                                                        :=
[X=]((.|<BR>)*"http://"(.|<BR>)*|(.|<BR>)*www(.|<BR>)*)('.|',|';)$   =>
[X]
    addPoint                                                        :=
[X=]((.|<BR>)*"http://"(.|<BR>)*|(.|<BR>)*www(.|<BR>)*) => [X]
    addPoint  := [P=].[X=](.|<BR>)*[Sign=]('.|'?|'!)<SP>*$ => @(upper,
[P])[X][Sign]
    addPoint   :=   [P=].[X=][on,   lazy](.|<BR>)*<SP>*$   =>   @(upper,
[P])[X]'.
    work3  :=  (<SP>|<BR>)*([N=](1|2|3|4|5|6)'.<SP>*[A=](.|<BR>)*)[N1  =
N  +  1]<BR>[X=]([N1]'.<SP>*(.|<BR>)*)  =>  <BR>"    "[N]". "@(addPoint,
[A])@(work3, [X])
    work3   :=   (<SP>|<BR>)*([N=](2|3|4|5|6)'.<SP>*[A=](.|<BR>)*)   =>
<BR>"   "[N]". "@(addPoint, [A])
    work3 := [X=].*'.<BR>[Y=](.|<BR>)+ => <BR>"   1. "[X]@(work4, [Y],
2)<BR>
    work3 := [X=].*<BR>[Y=](.|<BR>)+ => <BR>"    1. "[X]@(work4, [Y],
2)<BR>
    work3 := [X=].* => <BR>@(addPoint, [X])<BR>
    work4  :=  [X=].*'.<BR>[Y=](.|<BR>)+,  [N=].*  =>  <BR>"     "[N]".
"[X][N1 = N + 1]@(work4, [Y], [N1])
    work4 := [X=].*<BR>[Y=](.|<BR>)+, [N=].* => <BR>"   "[N]". "[X][N1
= N + 1]@(work4, [Y], [N1])
    work4 := [X=].*, [N=].* => <BR>"   "[N]". "@(addPoint, [X])
    work := (<SP>|<BR>)*([N=](1|2|3|4|5|6)'.<SP>*[A=](.|<BR>)*)[N1 = N
+    1]<BR>[X=]([N1]'.<SP>*(.|<BR>)*)    =>    <BR>[N]".    "@(addPoint,
[A])@(work, [X])
```

```
    work   :=   (<SP>|<BR>)*([N=](2|3|4|5|6)'.<SP>*[A=](.|<BR>)*)   =>
<BR>[N]". "@(addPoint, [A])
    work := [X=](.|<BR>)* => <BR>@(addPoint, [X])
```

4. Изменение нумерации вопросов. В каждом туре вопросы должны нумероваться с единицы.

```
    <Number> := (0|1|2|3|4|5|6|7|8|9)+
    <Tour> := <BR>"Тур:"<BR><Number>" тур"
    <Q> := <BR>"Вопрос "<Number>':<BR>
    [A=]<Tour>[source, S] => [A]@(recount, [S], 1)
    recount       :=       [A=][on,       lazy](.|<BR>)*<Q>[T=][on,
lazy](.|<BR>)+[cont](<Q>[Q=1]|<Tour>),    [Ind=]_([Q=0]|[source,    S,
0][Ind2=Ind+1][R=@(recount,    S,    Ind2)])    =>    [A]<BR>"Вопрос
"[Ind]':<BR>[T][R]
    recount := [A=][on, lazy](.|<BR>)*<Q>[T=][on, lazy]_, [Ind=]_ =>
[A]<BR>"Вопрос "[Ind]':<BR>[T]
```

5. Удаление лишних пустых строк между полями вопроса (пустая строка остаётся только перед ответом):

```
    <Number> := (0|1|2|3|4|5|6|7|8|9)+
    [A=](<BR>"Вопрос
"<Number>":"<BR>)[B=]([on,lazy](.|<BR>)*<BR>)[C=]("Автор:"<BR>)      =>
[A][on, full]@(clearbr, [B])[C]
    clearbr := <BR>+"Ответ:"<BR> => <BR><BR>"Ответ:"<BR>
    clearbr := <BR>+ => <BR>
```

6. Текст разбивается на строки, не превышающая длиной 72 символа каждая.

```
    [X=].* => @(cut, [X])
    cut  :=  [T=]([X=].*[@(length,  X)  <  73]<SP>[Y=].*)[@(length,  T)  >
72] => [X]<BR>@(cut, [Y])
    cut := [X=].* => [X]
```

В итоге, поставленная задача была решена, преобразование выполнялось за приемлемое время, а приложение предоставило пользователю возможность редактирования правил без его перекомпиляции и выключения лишних преобразований.

Извлечение данных (12)

Одной из программ автора является SIQuester – редактор вопросов для игры «Своя игра» (СИ). Он позволяет создавать пакеты вопросов для проведения викторины, используя программу в качестве ведущего. С использованием SIQuester’а возникала только одна проблема – большинство пакетов СИ были опять-таки написаны разными авторами, а потому представляли собой плоские файлы совершенно различного формата. Более того, поскольку пакеты часто были сборными, формат мог меняться несколько раз по ходу одного файла. Возникла естественная необходимость создать модуль импорта,

преобразовывавший бы такие файлы в формализованное представление пакета вопросов СИ, содержащего раунды, темы, вопросы и ответы на них, а также информацию об авторах, источниках и комментарии к различным элементами пакета.

В данной ситуации возникла задача извлечения данных из текстового файла. Извлечение данных – это также преобразование, на входе которого поступает текст, а на выходе получается некоторая структура данных, содержащая извлечённую информацию. Сформированная структура данных затем используется для дальнейшей работы.

Для выполнения процедуры извлечения данных программе необходимо знать формат, в котором эти данные хранятся. Используя описание этого формата, она и сможет читать текст, сравнивать его с форматом и в нужных местах «выписать» помеченные в формате элементы.

Для SPARD-преобразователя формат – это обыкновенный SPARD-шаблон, в котором переменные помечаются атрибутом `[on,m]` (режим записи совпадений). Переменные, получившие своё значение под действием этого режима, сохраняют его также как совпадение. Если совпадение было получено при проверке шаблона множества, оно передаётся вверх по процессу распознавания до тех пор, пока не будет возвращено в составе результата преобразования.

В результате извлечения данных SPARD-преобразователь может вернуть или иерархическую структуру, узлами которой являются имена множеств, содержащих совпадения, или линейный список таких совпадений. Пример ниже пояснит принцип работы преобразователя.

Авторские файлы вопросов используют разные форматы, поэтому для каждого из них требуется свой собственный шаблон SPARD. Поскольку хотелось максимально возможным образом облегчить процесс распознавания вопросов, задача автоматической генерации шаблона также была передана приложению. Приложение использует определённую эвристику (заключающуюся, например, в том, что любой вопрос начинается с новой строки и со стоимости, причём стоимость каждого последующего вопроса возрастает на одно и то же число) и разбивает текст на отдельные вопросы. Далее вопросы сравниваются между собой, определяются общие части и наиболее

длинные отличающиеся промежутки. Благодаря этому формируются неизменяемые символы шаблона и места, в которых будут искаться извлекаемые данные.

Рассмотрим пример того, как приложение построило шаблон для входного файла.

На вход преобразователю был подан файл с текстом:

```
Предварительные бои
Тема 1.     ...мор...
Вера Костягина
10.   Так принято называть хилое, тщедушное, физически недоразвитое существо
              заморыш
20.   При дворе этого правителя Милана Леонардо да Винчи исполнял обязанности военного инженера, гидротехника, устроителя придворных праздников
              Лодовико Моро
30.   Из песни Оскара Фельцмана известно, что это вовсе не птица, а сердце мальчишки, взлетевшее ввысь
              зимородок
40.   Этот генерал, командующий Рейнской армией демонстративно отказался от ордена Почётного Ле-гиона, назвав его "орденом почётной кастрюли"
              (Жан-Виктор) Моро
50.   Этот город в республике Татарстан обязан своим возникновением залежам медной руды, обнару-женным в XVII веке.
              Кукмор
Тема 2.     Животные в кино
Вера Костягина
10.   Собака этой породы – главный герой фильма "K-9"
              немецкая овчарка
20.   Именно он периодически наводит порядок в сказочной стране Нарнии
              (лев) Аслан
30.   В этом телесериале рассказывается об украденной корове, которую в целях сокрытия преступле-ния обули в валенки
              "Место встречи изменить нельзя"
40.   В этом фильме рассказывается о судьбе и душевной драме старого колхозника Танабая сквозь призму его непростых взаимоотношений с племенным жеребцом
              "Прощай, Гульсары"
50.   Эта картина фабрики Акционерного общества "А. Ханжонков и Ко" в 1913 году была поднесена це-саревичу Алексею, за что общество было удостоено высочайшей благодарности и наград
              "Стрекоза и муравей"
Тема 3.     Зеркала
Владимир Коляда
10.   Именно она написала роман "Зеркало треснуло"
      Агата Кристи
```

и т.д.

В результате автоматического разбиения файла на вопросы и последующего построения шаблона получилось:

Шаблон темы:

```
    <Line>Тема<Some>".     "[on,m]([TName                                =
]<TName>)<Line>[on,m]([TAuthor   =   ]<TAuthor>)('([on,m]([TComment   =
]<TComment>)'))?
```

Шаблон вопроса:

```
    <Line>[on,m]([Number   =   ]<Number>)[on,ignoresp]0'.[on,m]([Text   =
]<Text>)<Line>' [on,m]([Answer  =  ]<Answer>)([on,ignoresp]Комментарий":
"[on,m]([QComment        =        ]<QComment>))?([on,ignoresp]Источник":
"[on,m]([QSource    =    ]<QSource>))?([on,ignoresp]Автор'    вопроса":
"[on,m]([QAuthor = ]<QAuthor>))?
```

Программа содержит встроенный редактор SPARD, поэтому пользователь видит эти правила в более вменяемом формате и может их подправить.

Переменные <Number>, <Text>, <Answer> и т.д. содержат места, в которых будут располагаться интересующие нас данные. Можно увидеть, что за номером вопроса требуется наличия символов «0.», затем должен идти текст вопроса и на отдельной строке после пробелов – ответ. Тема после слова «тема» содержит изменяющуюся, но не интересующую нас информацию (<Some>), затем точку и символ табуляции, а потом – название темы.

Таким образом, преобразователь готов к импорту, заключающемуся в последовательном чтении каждого из вопросов и созданию объектов с извлечёнными свойствами в памяти. Общий шаблон для распознавания вопроса выглядит так:

```
    <Body> := [on,keepinitiator]<Q><SP>*$
    <Q> := [on,keepinitiator]<Q0>
    <Q0>      :=      [@(log,      q,      0)]<Line>[on,m]([Number      =
]<Number>)[on,ignoresp]0'.[on,m]([Text         =         ]<Text>)<Line>'
     [on,m]([Answer       =       ]<Answer>)([on,ignoresp]Комментарий":
"[on,m]([QComment        =        ]<QComment>))?([on,ignoresp]Источник":
"[on,m]([QSource    =    ]<QSource>))?([on,ignoresp]Автор'    вопроса":
"[on,m]([QAuthor = ]<QAuthor>))?
    <T0> := [@(log, t, 0)]<Line>Тема<Some>".    "[on,m]([TName        =
]<TName>)<Line>[on,m]([TAuthor   =   ]<TAuthor>)('([on,m]([TComment   =
]<TComment>)'))?
    <T> := [on,keepinitiator]<T0>
    <R0> := [@(log, r, 0)]<Line><Line>[on,m]([RName = ]<RName>)
```

```
<R> := [on,keepinitiator]<R0>
<RName> := (.|<BR>)+
<TName> := [on,lazy](.|<BR>)+
<TAuthor> := [on,lazy].+
<TComment> := [on,lazy].+
<Number> := <Digit>+
<Digit> := 0|1|2|3|4|5|6|7|8|9
<Text> := [on,lazy](.|<BR>)*
<Answer> := [on,lazy].+
<QComment> := [on,lazy](.|<BR>)+
<QSource> := [on,lazy](.|<BR>)+
<QAuthor> := [on,lazy](.|<BR>)+
<Some> := [on,lazy].*
<Line> := <SP>*[on,lazy](<BR><SP>*)+
[on,keepinitiator]<Body>[lmatches] => @matches
```

Результат преобразования получается через вызов функции `@matches`, которая, собственно, и выдаёт все совпадения в выходной поток. В результате чтения с помощью такого шаблона вопроса:

```
10. Так принято называть хилое, тщедушное, физически недоразвитое существо
                заморыш
```

получились следующие совпадения:

Number	1
Text	Так принято называть хилое, тщедушное, физически недоразвитое существо
Answer	заморыш

В результате анализа создаётся объект вопроса, у которого заполняются поля стоимости, текста вопроса и текста ответа.

В процессе анализа последовательно считываются все вопросы, и на выходе получается полностью структурированный вопросный файл.

Ещё одной проблемой преобразования является наличие ошибок во входном файле. Автор не везде выдерживает свой стиль, поэтому часто преобразователь не может успешно распознать вопрос. При возникновении этой ситуации пользователю выдаётся окно, в котором он может или заменить шаблон, или исправить входной текст. В случае замены шаблона впоследствии используются и старый, и новый шаблон, что обеспечивает возможность распознавания файлов с различными стилями оформления.

Работа с естественным языком

Последний пример являет собой лишь концепцию, а не завершённую задачу.

SPARD изначально разрабатывался для преобразования текстов на естественном языке, т.е. текстов, лишённых формальной структуры. Поэтому он легко может быть приспособлен для решения задачи анализа естественноязыковых текстов и построения на их основе формальных высказываний и обратной задачи – синтеза текстов из формальных структур.

Работа с естественным языком обычно представляет собой решение двух задач:

1. Преобразование текстов на естественном языке в формальные конструкции (анализ).
2. Обратное преобразование формализмов в текст (синтез).

Обе задачи – анализ и синтез – могут быть решены посредством выполнения последовательности более простых шагов. Каждый шаг представляет собой перевод входных данных из одного представления (уровня) в другое. Выделяются следующие уровни текста:

1. Плоский текст.
2. Лексический уровень (последовательность лексем текста).
3. Морфологический уровень (последовательность начальных форм слов с их грамматическими характеристиками).
4. Синтаксический уровень (синтаксическое дерево предложения).
5. Семантический уровень (представление текста на формальном языке описания смысла).

Таким образом, анализ и синтез состоят как минимум из 4 этапов преобразования (можно выделять и меньшие подэтапы). Этапы называются в соответствии с уровнями, которые они затрагивают. Например, преобразование из лексического уровня в морфологический называется морфологическим анализом, а обратный процесс – морфологическим синтезом.

Каждое из преобразований может быть описано на SPARD. Язык SPARD устроен так, чтобы правила преобразования можно было легко формировать.

В частности, синтаксическое дерево можно описать через множества SPARD:

```
<Предложение> := <Группа подлежащего><Группа сказуемого>
```

```
    <Группа подлежащего> := <Группа существительного, И>
    <Группа существительного, [Падеж]> := <Прилагательное, [Число],
[Род], [Падеж]>?<Существительное, [Число], [Род], [Падеж]>
    <Группа сказуемого> := <Сказуемое><Дополнение>?
    <Сказуемое> := <Глагол>
    <Дополнение> := <Группа существительного, [Падеж]>
    [match]<Предложение> => [matches]
```

И т. д.

Унификация обеспечивает совпадение числа, рода и падежа у существительного и подчинённого ему прилагательного. SPARD-преобразователь обеспечивает механизм разбора входных данных и строит иерархическую структуру предложения.

А вот правила, обеспечивающие лексический анализ и синтез для русского языка:

```
    <Letter>                                                        :=
а|б|в|г|д|е|ё|ж|з|и|й|к|л|м|н|о|п|р|с|т|у|ф|х|ц|ч|ш|щ|ъ|ы|ь|э|ю|я|А|Б|
В|Г|Д|Е|Ё|Ж|З|И|Й|К|Л|М|Н|О|П|Р|С|Т|У|Ф|Х|Ц|Ч|Ш|Щ|Ъ|Ы|Ь|Э|Ю|Я|''|'-
|0|1|2|3|4|5|6|7|8|9
    <Sign>                                  :=                      '—
|'/|'+|'%|'*|'=|'^|',|'.|'?|'!|'…|':|';|'«|'»|'"|'„|'“|'('|')|'['|']|'{|
'}|'<|'>|<BR>
    <Constant> := (.&!<SP>)+
    <Word> := <Letter>+
    <Before> := '—|'/|'+|'%|'*|'=|'^|'«|'"|'„|'(|'[|'{|'<
    <AllowSpace> := {Category:Sign, Value:<Before>} | {Category:Word,
Value:.+} | {Category:Constant, Value:.+}
    <After>                                 :=                      '—
|'/|'+|'%|'*|'=|'^|',|'.|'?|'!|'…|':|';|'»|'"|'“|')|']|'}|'>
    [T=]<Word> => {Category:Word, Value:[T]}
    [T=]<Sign> => {Category:Sign, Value:[T]}
    [T=]<Constant> => {Category:Constant, Value:[T]}
    [T]' <= {Category:Word|Constant, Value:[T=].+}[cont]<AllowSpace>
    [T] <= {Category:Word|Constant, Value:[T=].+}
    [T]' <= {Category:Sign, Value:[T=]<After>}[cont]<AllowSpace>
    [T] <= {Category:Sign, Value:[T=](.+|<BR>)}
```

Прямые правила преобразования (=>) обеспечивают анализ текста и построения последовательности объектов-лексем. Каждая из лексем содержит информацию о типе лексемы (слово, знак или константа) и значение лексемы (совпавший элемент текста).

Обратные правила преобразования (<=) обеспечивают синтез текста. Таким образом, один SPARD-преобразователь может выполнять оба преобразования.

Комбинируя преобразователи друг с другом, можно получить цепочку, обладающую способностью на основе входного текста строить формальное высказывание, имеющее

тот же смысл, что и текст. Данный преобразователь будет являться основным звеном в построении искусственного интеллекта.

Перспективы развития SPARD

Описанная версия языка является предварительной. И если основная концепция «шаблон – результат» уже устоялась и не претерпит изменений, в языке есть много мест, использование которых вызывает вопросы. В процессе дальнейшей проработки и внесения предложений возможно изменение спецификации языка до тех пор, пока он не достигнет приемлемого состояния.

Помимо этого, планируется развитие текущей версии SPARD-преобразователя. В настоящий момент преобразователь использует заданные правила «как есть», строя по каждому правилу дерево выражений и используя его для проверки входных данных. Данный подход приводит к необходимости многочисленных рекурсивных вызовов функции сравнения шаблона с входом (каждый узел дерева выражений вызывает подобную функцию у своих узлов-операндов) и к необходимости большого числа возвратов при чтении входного текста. Возврат возникает всякий раз, когда проверяется предикат с различными вариантами совпадений (например, предикат ИЛИ) и первоначально выбранный вариант совпадения не привёл к успеху.

Предполагается замена этой древовидной структуры таблицей переходов, в которой на основании состояния преобразователя и текущего входа выбирается новое состояние и (возможно) на выход выдаётся какой-то частичный результат. Преобразователь, работающий по такой таблице, будет осуществлять преобразование без рекурсивных вызовов и возвратов, и поэтому процесс преобразования значительно ускорится.

Процесс построения таблицы переходов на основе правил SPARD являет обобщением алгоритма построения преобразователя по заданной грамматике. Фактически, он представляет собой задачу построения LL-анализатора по packrat-анализатору.

Впоследствии таблица переходов может служить источником для генератора кода. На выходе такой генератор будет выдавать исходный код или готовое приложение, осуществляющее преобразование, которое изначально было задано правилам SPARD.

Список использованной литературы

1. Спецификация языка XSLT. [В Интернете] http://www.w3.org/TR/xslt.

2. **А. Ахо, Дж. Ульман.** *Теория синтаксического анализа, перевода и компиляции.* Москва : Мир, 1978.

3. Википедия. Описание языка РЕФАЛ. [В Интернете] http://ru.wikipedia.org/wiki/%D0%A0%D0%95%D0%A4%D0%90%D0%9B.

4. Regexp Grammars - расширение языка Perl. [В Интернете] http://search.cpan.org/~dconway/Regexp-Grammars-1.012/lib/Regexp/Grammars.pm.

5. Генераторы парсеров peg и leg. [В Интернете] http://piumarta.com/software/peg/peg.1.html.

6. Язык программирования AWK. [В Интернете] http://ru.wikipedia.org/wiki/AWK.

7. Утилита Chomski. [В Интернете] http://en.wikipedia.org/wiki/Chomski.

8. Язык программирования F#. [В Интернете] http://ru.wikipedia.org/wiki/F_Sharp.

9. Язык программирования TREE-META. [В Интернете] http://en.wikipedia.org/wiki/TREE-META.

10. Язык программирования COMIT. [В Интернете] http://www.mt-archive.info/MT-1958-Yngve.pdf.

11. Программа Vopros. [В Интернете] http://vladimirkhil.com/vopros/.

12. SIQuester: Редактор вопросов «Своей игры». [В Интернете] http://vladimirkhil.com/si/siquester/.

Ссылки по теме

1. Спецификация SPARD (http://vladimirkhil.com/lingware/spard/specification/SPARD.pdf).

2. Форум для обсуждения SPARD

(http://vladimirkhil.com/forum/viewforum.php?f=13).

3. Текущая реализация SPARD-преобразователя, доступная онлайн (http://vladimirkhil.com/lingware/spard/implementation).