УДК 004.434

СРЕДСТВА ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ ФОРМАЛЬНЫХ ГРАММАТИК И ИХ ПРИМЕНЕНИЕ ДЛЯ СОЗДАНИЯ ДИАЛЕКТОВ

А.А. Бреслав

Современные средства разработки конкретного синтаксиса предоставляют три основных механизма повторного использования: модули, шаблоны и аспекты. В работе рассмотрены эти механизмы и степень полноты их реализации в существующих инструментах, а также предложены их расширения и рассмотрено применение расширенных версий для создания диалектов языков программирования.
Ключевые слова: синтаксис, грамматика, шаблон, аспект, диалект.

Введение

Механизмы повторного использования во многом определяют пригодность средств разработки для промышленного программирования. С распространением предметно-ориентированных языков [1] вопросы повторного использования синтаксических определений (формальных грамматик и формализмов на их основе) встали особенно остро, поскольку традиционные средства разработки синтаксиса [2–4] слабо его поддерживали. Более современные инструменты сильно продвинулись в этом смысле.

В настоящей работе обсуждаются используемые ныне механизмы повторного использования синтаксических определений и возможности их улучшения.

Одним из наиболее мощных современных инструментов для генерации синтаксических анализаторов является SDF [5]. Эта система поддерживает модульные определения и параметризованные модули. Параметрами модуля являются символы грамматики, что позволяет создавать правила, использующие не заранее зафиксированные символы, а значения параметров.

Не менее мощный инструмент Rats! [6] также поддерживает параметризованные модули, но параметрами могут быть лишь другие модули, а не символы, как в SDF, таким образом в Rats! можно варьировать лишь наборы символов, используемые модулем, а не отдельные символы. Кроме этого, Rats! позволяет модифицировать существующие правила, добавляя, удаляя или заменяя отдельные продукции.

Несколько иной подход выбран в генераторе компиляторов LISA [7]. Этот инструмент поддерживает наследование грамматик без параметризации. Шаблоны (параметризованные определения) используют только задания семантических действий. Кроме того, для присоединения семантических действий к грамматике используются аспекты в интерпретации, сходной с AspectJ [8].

Как легко заметить, так или иначе используются три механизма: модули (повторно используемые наборы правил), шаблоны (параметризованные определения) и аспекты (изменения или дополнения грамматики, определенные вне ее). Наследование грамматик в LISA по сути эквивалентно подключению модулей с небольшими расширениями, которые обобщаются при помощи параметризации отдельными символами. Модификации грамматики в Rats! – это упрощенные аспекты. Легко заметить и то, что два более сложных механизма – шаблоны и аспекты – используются в очень ограниченной версии: параметризация – только по символам или модулям, аспекты – только на уровне всей продукции.

Предлагается подход, использующий полноценные шаблоны и аспекты для повторного использования грамматик. Кратко описывается система Grammatic [9], в рамках которой этот подход реализуется. Подробно рассматриваются возможности полноценных шаблонов и аспектов, а затем демонстрируется их использование для создания диалектов существующих языков без написания кода на низком уровне абстракции.

Система разработки текстового синтаксиса Grammatic

Система Grammatic разрабатывается как инструмент, позволяющий манипулировать синтаксическими определениями на высоком уровне абстракции и применять другие средства разработки (такие, например, как генераторы синтаксических анализаторов) к этим определениям.

Основу системы составляет язык описания грамматик, основанный на расширенной форме Бэкуса–Наура (Extended Backus–Naur Form, EBNF) и использующий нотацию, принятую во многих инструментах:

```
rule : NAME metadata? ':' production ('||' production)* ';' ;
```

Основной особенностью этого языка является возможность аннотирования грамматики произвольными метаданными. Метаданные позволяют сопровождать синтаксическое определение семантической информацией, которая требуется сторонним инструментам. Например, в качестве метаданных могут быть описаны семантические действия, предикаты или декларативные средства разрешения конфликтов, правила обратного преобразования дерева разбора в текст, правила расстановки переводов строк и пробелов и т. д. Вот пример определения символа с сопоставленными ему метаданными (указаны в фигурных скобках):

```
COMMENT {lexical; channel=hidden} : '//' ([^'\n'])* '\n';
```

Как видно из примера, метаданные представляют собой набор именованных атрибутов, которые могут иметь значения различных типов или вовсе не иметь значения (в этом случае учитывается лишь факт наличия самого атрибута).

Модули и шаблоны

Grammatic поддерживает модули аналогично SDF или Rats!, т.е. позволяет описывать модуль в отдельном файле и подключать его с помощью предложения import. Основное отличие Grammatic состоит в понимании концепции шаблонов, а именно в том, что параметром (и результатом применения) шаблона может быть не только модуль или символ, а любой объект, который может упоминаться в определении грамматики – часть синтаксического описания, метаданные, перечисление символов и т.д.

Для примера возьмем конкретный синтаксис для метаданных самого Grammatic. В его определение входят правила для различных типов значений атрибутов, правило для значения атрибута вообще задается таким шаблоном:

```
Symbol template attributeValue<Production* other> {
 attributeValue   : character
      | INT
      | STRING
      | NAME
      | metadata
      | '{{' term* '}}'
      | <other> ;
}
```

В базовом определении этот шаблон инстанцируется так (параметр other имеет значение пустого набора продукций):

```
import attributeValue<>;
```

Допустим, что в какой-то грамматике, использующей это определение, нужно добавить продукцию, позволяющую использовать символы грамматики в виде значений метаданных. Это легко сделать, передав в качестве параметра шаблона эту продукцию:

```
import attributeValue<| '#' symbolReference>;
```

Как видно из примера, полноценные шаблоны позволяют реализовать функциональность, которая реализована в других инструментах отдельным механизмом.

Аспекты

Многие инструменты используют языки, расширяющие EBNF специальным образом, например, к стандартной нотации добавляются типы возвращаемых значений и параметры для правил, код на языке общего назначения для семантических действий и т.д. Если эти расширения интенсивно используются, полученные определения становятся очень плохо читаемыми из-за смешения различных видов информации. В связи с этим возникает потребность выносить дополнительную информацию за пределы синтаксического определения, что улучшает модульность системы, разделяя синтаксис и прочие элементы определения.

В Grammatic все расширения языка определений выражаются метаданными, поэтому такое разделение сводится к вынесению метаданных за пределы определения грамматики. Это обеспечивается механизмом точек присоединения и дополнений (point-cut and advice, [8]), аналогичным используемому в AspectJ. Чтобы присоединить метаданные к грамматике, необходимо обозначить те объекты, к которым они присоединяются. Эта задача решается с помощью структурных запросов. Запрос регламентирует свойства объектов, которые необходимо найти, например, «найти символ, в определении которого есть продукция, начинающаяся с него самого»:

```
$S |> $S ..
```

К выбранным объектам (сохраненным в переменные) можно присоединять метаданные. По аналогии с AspectJ и другими аспектными языками наборы таких правил присоединения метаданных называются аспектами, поскольку каждый такой набор действительно отражает определенный аспект системы, улучшая тем самым логическое разделение информации.

Важным свойством аспектов в Grammatic является то, что они могут быть не привязаны к конкретным правилам грамматики: запросы выражают общие свойства объектов, поэтому их можно применять в любом контексте.

Проиллюстрируем использование аспектов (и шаблонов) для решения задачи создания диалектов.

Создание диалектов

Диалекты, т.е. языки, немного модифицирующие синтаксис (и иногда – семантику) других языков, довольно широко распространены. Так, Java 5 во многом представляет собой диалект Java 2, практически каждый генератор синтаксических анализаторов использует свой диалект EBNF, а каждая СУБД – свой диалект SQL. Будем рассматривать диалекты, не привносящие новых семантических единиц, т.е. по сути вводящие так называемый «syntax sugar». Нередко этого достаточно, особенно если исходный язык спроектирован для создания диалектов (например, в языке предусмотрены метаданные достаточно произвольной структуры). Как нетрудно заметить, язык определений Grammatic спроектирован как раз таким образом, что позволит нам легко специализировать его, делая его использование незаметным для пользователей, привыкших к другим инструментам. Для этих целей используются аспекты.

В качестве примера рассмотрим создание диалекта Grammatic, имитирующего синтаксис генератора ANTLR [3] – одного из наиболее популярных на сегодняшний день среди программистов на Java. Определение символа в ANTLR выглядит так:

```
qualifiedName [SymbolTable table] returns [Variable result]
 : objectName=name '.' variableName=name
 {
 result = table.lookUp(objectName, variableName);
 };
```

Нетрудно видеть, что оно заметно отличается от определения в Grammatic: добавлены формальные параметры и тип возвращаемого значения, имена для ссылок в правой части и код семантического действия. Несмотря на то, что смешение различных типов информации считается вредным, мы, тем не менее, опишем здесь способ заставить Grammatic вести себя так же, чтобы продемонстрировать удобство создания диалектов.

Итак, рассмотрим определение в Grammatic, аналогичное приведенному выше (вся дополнительная информация вынесена в метаданные):

```
qualifiedName {
 parameters = {{{type = 'SymbolTable'; name = 'table'}} ;
 returns = {type = 'Variable'; name = 'result';} ;
}: name{variable = 'objectName'} '.' name {variable = 'variableName'}
 @{
 after = 'result = table.lookUp(objectName, variableName);';
 };
```

Метаданные позволяют легко выразить те же понятия, однако синтаксис отличается. Для решения таких задач Grammatic позволяет создавать специальные аспекты, модифицирующие синтаксис и сопоставляющие таким модификациям семантику в терминах исходного языка. Во время выполнения новые синтаксические конструкции просто транслируются в метаданные. Начнем с простого – поддержим имена переменных:

```
atom |> $symbolName:NAME
      [[
            $symbolName.before = {
                  syntax = <<(NAME '=')?>>;
                  transformTo = {{$symbolName.variable := NAME;}};
            };
      ]];
```

Это правило из обыкновенного аспекта, сопоставляющее элементам определения метаданные. Выбрана точка расширения – использование символа NAME в качестве ссылки на символ, и ей сопоставлен атрибут before, определяющий модификацию синтаксиса, добавляющую конструкцию перед ним. Атрибут syntax содержит описание добавляемой конструкции, оно соответствует префиксу для символа в ANTLR. Атрибут transformTo описывает преобразование добавленной синтаксической конструкции в метаданные: во время выполнения будет создан атрибут variable, и ему присвоится значение токена NAME.

Эти метаданные сами по себе ничего не значат, чтобы они «заработали», необходима программа, которая их интерпретирует. Такой интерпретатор входит в Grammatic – он отвечает за модификацию определений с помощью аспектов.

Продолжим создавать наш диалект. Добавим возможность писать в конце продукции семантические действия:

```
production --> $pre:('@' metadata)? alternative $post:('@' metadata)?
      [[
            $pre.instead = { syntax = << >>; };
            $post.instead = {
                  syntax = << $code:STRING_IN_CURLY >>;
                  transformTo = {{
                        alternative.after := $code;
                  }};
            };
      ]];
```

Здесь выбраны две точки расширения – метаданные перед и после продукции, и для обеих используется замещение, т.е. точка расширения заменяется новой синтаксической конструкцией. В первом случае замещающая конструкция пуста, что соответствует удалению части исходного правила. Во втором случае использован токен «строка

в фигурных скобках», соответствующий синтаксису определяемого расширения. Во время выполнения значение этой строки будет записано в атрибут after.

Последнее расширение будет наиболее многословным, поскольку для него придется определить два новых правила – parameter и parameters, определяющие синтаксис параметра и списка параметров в ANTLR (сами эти определения опущены для краткости):

```
rule
 --> $ruleName:NAME metadata? ':' production ('||' production)* ';'
 [[
 metadata.instead = { syntax = << >>; };
      $ruleName.after = {
       syntax = <<$p:parameters? ('returns' '['$r:parameter']')?>>;
       transformTo = {{
            parameters := $p;
            result := $r;
       }}
      };
 ]];
```

Произвольные метаданные для символа удаляются, зато после имени символа добавляются определения параметров и возвращаемого значения, транслируемые в соответствующие атрибуты.

Этот механизм полностью реализует идеи AOP, перенесенные на почву формальных грамматик. Он позволяет достичь большой гибкости и значительно расширяет возможности разработчика по сравнению с упомянутыми выше инструментами.

Заключение

Рассмотрены современные подходы к повторному использованию формальных грамматик: модули, шаблоны и аспекты. Предложены обобщения этих подходов, существенно расширяющие их возможности по сравнению с существующими реализациями.

Предложено использование шаблонов с параметрами произвольных типов и полноценные аспекты, основанные на точках присоединения и дополнениях. Предложенные подходы реализуются в системе Grammatic, разработанной автором.

Показано, что использование предложенных обобщений позволяет не только полностью реализовать функциональность, доступную в существующих инструментах, но и получить принципиально новые возможности. В частности, полноценные аспекты позволяют создавать диалекты существующих языков, не требующие дополнения их семантической структуры, без вмешательства в код семантического анализатора, что как правило, не достигается при других подходах.

Механизм создания диалектов проиллюстрирован на примере самой системы Grammatic, язык определений которой может быть специализирован для упрощения работы с метаданными определенной структуры.

Работа выполнена при финансовой поддержке в форме гранта Правительства Санкт-Петербурга № 3.11/4-05/55.

Литература

1. Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai, G. Composing domain-specific design environments // Computer. – 2001. – Vol. 34. – № 11. – С. 44–51.
2. Johnson S.C. Yacc: Yet another compiler compiler – UNIX Programmer’s Manual, Vol. 2. – Holt, Rinehart, and Winston, New York, 1979. – P. 353–387.

3. Parr T. The definitive ANTLR reference // The Pragmatic Bookshelf, 2007.
4. Gagnon E., Hedren L. SableCC, an Object-Oriented Compiler Framework // Technology of Object-Oriented Languages and Systems, 1998.
5. Klint P.A. Meta-Environment for generating programming environments // ACM Transactions on Software Engineering and Methodology. – 1993. – Vol. 2. – № 2. – P. 176–201.
6. Grimm R. Better extensibility through modular syntax – ACM PLDI ’06, New York, 2006. – P. 38–51.
7. Gagnon E., Hendren L. An object-oriented compiler framework // TOOLS, 1998. – P. 140–154.
8. Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G. An overview of AspectJ // Proceedings of the 15th European Conference on Object-Oriented Programming. – London, Springer-Verlag, 2001. – P. 327– 53.
9. Бреслав А.А., Попов И.Ю. Применение принципов MDD и аспектно-ориентированного программирования к разработке ПО, связанного с формальными грамматиками // Научно-технический вестник СПбГУ ИТМО. – 2008. – Вып. 57. – С. 87–97.

Бреслав Андрей Андреевич – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, аспирант, abreslav@gmail.com