ПРИМЕНЕНИЕ ПРИНЦИПОВ MDD И АСПЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ К РАЗРАБОТКЕ ПО, СВЯЗАННОГО С ФОРМАЛЬНЫМИ ГРАММАТИКАМИ

А. А. Бреслав (СПбГУ ИТМО)

Научный руководитель — д. ф.-м. н., профессор И. Ю. Попов

Современные средства разработки грамматик и связанных с ними приложений не удовлетворяют принципам инженерии ПО: различные аспекты системы смешиваются, нарушается модульность и разграничение уровней абстракции. В настоящей работе представлена концепция средства разработки, призванного решить эти проблемы, используя принципы модельно-ориентированной разработки и аспектно-ориентированного программирования.

Введение

Разрабатывая программные (и не только программные) системы, необходимо следовать определенным правилам, чтобы совладать со сложностью, присущей решаемой задаче [1]. Эти правила вырабатывались поколениями разработчиков, и ныне они являются основой современного программирования. Речь идет о поддержании на должном уровне так называемых *факторов внутреннего качества* [2], то есть тех свойств системы, которые, будучи не видны для конечного пользователя, в значительной мере обеспечивают уровень качества *внешнего*, то есть заметного простому потребителю: корректность, стабильность, удобство в использовании и т.д.. К факторам внутреннего качества относят, в числе прочих, и следующие показатели:

- понятность программного кода,
- модульность и повторное использование модулей,
- разделение уровней абстракции (слоев) в системе,
- расширяемость.

В настоящей работе мы обратимся к тем программным системам, которые так или иначе сильно зависят от некоторых формальных языков и их грамматик (в работе [3] для таких систем вводится довольно удачное на наш взгляд название *grammarware*). К этому классу программных систем относятся, например, компиляторы, интерпретаторы, средства статического анализа программного кода, генераторы кода, средства трансформации программ и другие инструменты, используемые для работы с программным кодом. Кроме того, многие приложения используют собственные «малые» языки для файлов конфигурации (в последнее время для этой цели используют XML, грамматикой для которого являются XSD или DTD-определения). В последние годы стали популярны модельно-ориентированные подходы [4], а с ними и предметно-ориентированные языки (с текстовым или графическим синтаксисом), ставшие инструментами уже не только для программистов, но и для конечных пользователей [5].

Все это многообразие приложений страдает от одной серьезной проблемы: внутреннее качество оставляет желать много лучшего в той части, где речь идет о грамматиках и связанных с ними артефактах разработки. Хорошо развитые в других областях программирования подходы и инструменты, обеспечивающие поддержание на должном уровне факторов внутреннего качества системы, попросту не применяются для grammarware [3]. Написанные вручную или с использованием генераторов синтаксические анализаторы и другие средства обработки языков как правило монолитны (разбиение на модули редко заходит дальше разделения лексического и синтаксического анализа), то есть происходит смещение различных аспектов системы (и уровней абстракции). Как следствие такие программы трудно читать, поддерживать и

развивать, а использовать такой код повторно почти невозможно.

В числе основных причин этого явления называют недостаточное понимание проблемы разработчиками и спад интереса к теории формальных грамматик [3] , а также сложности, возникающие при переносе концепций (таких как, например, наследование и полиморфизм) на грамматики [6].

Справедливости ради стоит отметить, что в современных средствах автоматизации разработки grammarware предпринимаются попытки решить означенные проблемы. Так генератор синтаксических анализаторов SableCC разделяет определение грамматики и семантические действия, используя схему трансляции [7], система ANTLR предоставляет возможность повторного использования ряда артефактов за счет наследования грамматик и анализаторов деревьев (Tree Parsers) [6]. Кроме того, существуют системы, строящие на выходе готовые приложения (xText [8], TCS [9]), полностью генерируемые, а потому не подлежащие ни чтению, ни расширению вручную.

Однако этих достижений недостаточно: традиционные генераторы, как бы гибки они ни были, решают низкоуровневую задачу построения синтаксического анализатора по грамматике определенного класса, что существенно ограничивает возможности разработчика в смысле разделения системы на модули и слои, а системам, строящим готовые приложения, не хватает гибкости — они используются в основном для прототипирования.

Существуют и средства, основанные на формальном подходе к определению языков (речь как правило идет о формальном определении семантики, но общая «научность» подхода распространяется и на синтаксис). Такие системы [10, 11] более пригодны для построения универсальных инструментов проектирования, но они создаются учеными-теоретиками преимущественно в исследовательских целях и поэтому (по крайней мере пока) не вполне пригодны для использования в индустрии.

В настоящей работе мы представляем концепцию инструментальной платформы, которая призвана обеспечить разработчиков grammarware аппаратом, позволяющим поддерживать внутреннее качество их продуктов на должном уровне. Мы стремимся обеспечить разделение различных аспектов системы и различных уровней абстракции, а также облегчить эволюционные изменений как системы в целом, так и отдельных модулей.

Предлагаемый подход основан современных концепциях разработки ПО: MDD (Model Driven Development [4]) и аспектно-ориентированном [12] и генеративном программировании [13].

В нашей системе контекстно-свободная грамматика задается в EBNF [14] без каких-либо априорных ограничений на класс грамматики. Определение грамматики может быть дополнено метаданными произвольной структуры. Элементы грамматики, выбираемые специализированными запросами, становятся входными данными для различных преобразований. Результатами преобразований являются всевозможные артефакты разработки: новые варианты грамматики, различные модели или код. Также поддерживается возможность использования какого-либо генератора синтаксических анализаторов.

В следующих разделах дается подробное описание подхода, после чего представлен пример его использования для создания расширяемого анализатора арифметических выражений.

Определение грамматики

Как уже было сказано, грамматика в нашей системе задается в расширенной форме Бэкуса-Наура (Extended BNF, EBNF [14]), причем мы будем использовать

нотацию, проиллюстрированную ниже:

```
Rule -> NontermSpec '->' {SymbolSpec} ';' ;
```

Здесь символ фигурные скобки обозначают ноль или более повторений , знак '->' разделяет левую и правую части правила. Имена нетерминальных символов начинаются с заглавной буквы, простые терминалы задаются в одинарных кавычках. Вот еще один пример:

```
Terminal -> id | quotedCharacters;
AttributeList -> Attribute {',' Attribute} ;
```

Именованные терминалы называются со строчной буквы, а знак '|' обозначает альтернативу. Прочие обозначения будут поясняться по мере надобности.

Согласно идеологии MDD, мы рассматриваем определение грамматики не как текст на некотором языке, а как *модель*, то есть набор объектов, связанных ассоциациями [15]. По сути этот набор объектов есть ни что иное как абстрактное синтаксическое дерево (AST [16]) определения грамматики. Далее мы будем говорить об *объектах* или *элементах* грамматики, имея в виду элементы этого дерева, то есть смысловые единицы описания грамматики. Примерами таких единиц будут символы: нетерминалы, и терминалы, правила, отдельные альтернативы и т.д.

Априори на грамматику не накладывается никаких ограничений, то есть она может быть, например, сколь угодно неоднозначной, если это устраивает разработчика. Сама по себе система работает с грамматикой как с моделью и не пользуется семантикой ее определения.

Преобразования грамматики

Grammarware — это программное обеспечение, основанное на грамматиках, поэтому весь процесс разработки этого класса программных систем концентрируется вокруг определений различных грамматик, и состоит в расширении и анализе этих определений, а также в построении на их основе других артефактов разработки.

Располагая определением синтаксической структуры языка и соответствующими инструментами для работы с ним, разработчик может сделать очень многое. Например (самый примитивный случай), можно автоматически преобразовать исходное определение к классу LL(*) и сгенерировать определение синтаксического анализатора для ANTLR.

Кроме собственно синтаксической структуры языка, разработчик может извлекать из грамматики и другие виды информации, например, построить Java-классы вершин AST разрабатываемого языка и внедрить в генерируемое ANTLR-определение код построения AST. Существует масса других полезных приложений, значительная часть кода которых может быть получена на автоматически на основе грамматик: трансляция одного языка в другой, определение правил форматирования кода, подсчет различных метрик и статический анализ кода и т.д.

Наша задача — предоставить необходимый инструментарий для подобных операций с грамматиками. Все преобразования задаются с помощью специализированных языков, являющихся частью нашей системы. Каждое преобразование может быть повторно использовано.

Виды преобразований

Преобразования (или *трансформации*) являются основными модулями при работе в рамках MDD. Фактически MDD состоит в том, чтобы запротоколировать весь процесс разработки в виде трансформаций, чтобы иметь возможность автоматически выполнить его от начала до конца в любой момент, если это понадобится: например, если изменятся входные данные, которые, как правило, отражают требования, или если в одной из трансформаций будет обнаружена ошибка.

Для работы с грамматиками в рамках предлагаемого подхода будут использоваться следующие виды преобразований:

- построение новой грамматики (или расширение существующей),
- построение экземпляра некоторой метамодели [17],
- генерация произвольного текста (например, кода на языке программирования),
- генерация определения грамматики на входном языке какого-либо программного средства (например, ANTLR или xText).

Возможность построения одной грамматики по другой — очень важный момент в жизненном цикле grammarware, поскольку таким образом можно обеспечить относительно безболезненную эволюцию системы, что в настоящее время является основной проблемой разработчиков подобных продуктов. Не менее важно уметь строить по грамматике различные модели, например, иерархии классов UML [15] или Ecore [18] или другие специализированные модели, на основе которых в рамках MDD будут генерироваться прочие артефакты разработки.

Генерация текста занимает особое место среди возможных преобразований, ее можно сравнить с использованием динамически типизированных языков программирования: как Ruby используется там, где Java слишком сложна и недостаточно гибка, так и простой текст заменяет строго регламентированные по структуре модели там, где их построение затруднено. Естественно, не нужно сбрасывать со счетов и необходимость построения программного кода — единственного представления программ, пригодного для дальнейшей компиляции и исполнения.

Последний вид преобразования, связанный с необходимостью использования существующих средств «генерации компиляторов», введен для удобства: теоретически его можно было реализовать как генерацию обычного текста, однако на практике создание таких генераторов — дело довольно трудное, к тому же популярных средств для работы с грамматиками не так уж много, и трансформации в их входные языки целесообразно включить в стандартную библиотеку. Такие преобразования будут своего рода «шаблонами»: всю механическую работу по преобразованию правил из нашей нотации нужный вид выполнит система (здесь речь не идет об изменении структуры правил — только о форме их записи), а код семантических действий (и другие расширения определения грамматики) будет определяться разработчиком.

Вне зависимости от вида преобразования разработчику потребуются несколько основных возможностей как то:

- обработка не всей грамматики целиком, а лишь некоторого набора объектов, выбранного по определенному критерию,
- отслеживание корректности входных данных и механизм сообщения об ошибках,
- взаимодействие различных преобразований.

Первая задача решается с помощью специализированного *языка запросов*, позволяющего определять критерии для выборки объектов из грамматики. Всякое преобразование работает с набором объектов, выбранным каким-то запросом. Подробнее язык запросов будет описан в следующем разделе.

Проверить корректность входных данных можно также с помощью языка запросов. Для очень широкого круга проверок вполне достаточно двух форм задания условия:

- множество объектов, выбранное запросом пусто, то есть в грамматике нет нужных объектов,
- или, наоборот, в грамматике существуют «нежелательные» объекты, удовлетворяющие определенному запросу, и множество, выбранное по этому запросу не пусто.

Как показано ниже, с помощью этих инструментов можно проверять довольно сложные условия, что обусловлено, конечно, мощностью языка запросов.

Для построения произвольных моделей и текста мы используем стандартные средства: вариацию QVT [19], позволяющую просто задать значения всех свойств каждого объекта, и язык текстовых шаблонов StringTemplate [20], для краткости мы не описываем здесь технические детали, связанные с ними.

Более подробно следует остановится на средствах построения грамматик. Мы стремимся сделать преобразование одной грамматики в другую максимально простым и удобным. Будем также иметь в виду, что нередко нужно не столько построить абсолютно новую грамматику, сколько несколько модифицировать имеющуюся, то есть большая часть новой грамматики будет скопирована из исходной.

Итак, некоторый запрос выбрал из исходной грамматики набор объектов (символов, правил или частей правил). Какие инструменты понадобятся разработчику трансформации, чтобы построить новую грамматику?

Основной операцией будет создание нового правила:

```
rule RuleDefinition -> 'rule' Nonterminal '->' RuleRightSide;
```

Эта строка создает новое правило и те символы в грамматике, которые входят в это определение. Кроме создания нового правила, нужно еще уметь добавлять альтернативы к существующим правилам. В нашей нотации это делается так:

```
rule RuleDefinition |> 'rule' Nonterminal '|>' RuleRightSide;
```

Символ «|>» означает, что то, что стоит от него справа, будет добавлено в определению нетерминала из левой части правила.

На самом деле это все, что нам понадобится (по крайней мере на данном этапе развития нашего понимания системы). Нужно только сказать, что объекты, выбранные запросом, можно встраивать в создаваемые правила. Например, если запрос вернул нетерминал N и цепочку символов грамматики definition (не важно, из какого она правла), можно создать такое правило:

```
rule N |> '(' definition ')';
```

Это будет означать, что мы добавили к определению нетерминала N в новой грамматике альтернативу, состоящая из цепочки символов definition, взятой в круглые скобки.

Для копирования правил из исходной грамматики в новую без изменений, можно воспользоваться таким трюком: пусть запрос выбрал нетерминал N и всю правую часть правила, которое его определяет RightSide, тогда достаточно написать следующее:

```
rule N -> RightSide;
```

Эта строка создаст правило, в левой части которого стоит нетерминал N, а в правой — правая часть правила, которое в исходной грамматике определяло N, то есть новое правило в точности совпадает со старым. В принципе можно обеспечить какой-нибудь более простой механизм для копирования правил без изменений, но рассмотрение этого вопроса выходит за рамки данной работы.

Теперь перейдем к определению языка запросов.

Язык запросов

Задача языка запросов — предоставить инструментарий для выбора объектов из грамматики по всевозможным критериям. К таким критериям относятся как индивидуальные свойства объектов (например, имя символа), так и *структурный контекст*: свойства фрагмента грамматики, частью которого является объект.

Вот примеры типичных запросов:

- нетерминал, имя которого оканчивается на «Expression»;
- правило, в правой части которого стоит ровно один символ;
- символ стоящий в каком-либо правиле между двумя нетерминалами;

- все альтернативы данного правила, кроме той, в которой есть символ X.

Настоящая работа не имеет целью исчерпывающее описание возможностей языка запросов (не все из которых в настоящий момент до конца разработаны), поэтому мы ограничимся лишь основными моментами.

Начнем с простых свойств выбираемых объектов. Нам понадобится выбирать символы грамматики по имени и типу (нетерминал, терминал). Для этих целей в языке запросов имеются следующие инструменты.

Запрос, представляющий из себя просто регулярное выражение для строки символов, с операторами «*» (любое количество любых символов) и «.» (один любой символ), определяет шаблон поиска по имени:

- `Expression` — символ с именем «Expression»;
- `*Expression` — символ, имя которого оканчивается на «Expression»;
- `.*Expression` — символ грамматики, имя которого заканчивается на «Expression», но содержит еще хотя бы один символ.

Для ограничения типа символа грамматики используются следующие обозначения (в квадратных скобках указывается ограничение на имя):

- `#`**`symbol`**`[Plus]` — любой символ (терминал или нетерминал) с именем «Plus»;
- `#`**`nonterm`**`[Expr*]` — нетерминал, имя которого начинается на «Expr»;
- `#`**`term`**`[*.num]` — терминал, имя которого заканчивается на num, но содержит еще хотя бы один символ.

Перейдем к структурному контексту. Положение того или иного символа в правиле и вид этого правила задается регулярным выражением на символах грамматики. В таких выражениях целесообразно использовать переменные (имя переменной предваряет вырадение для символа и отделяется от него двоеточием):

- `N -> x | N x` — в точности такое правило: «`N -> x | N x`»;
- `N:#`**`nonterm`** `-> x | N x` — правила такого вида, где в первой альтернативе стоит символ с именем «x», а во второй — он же, предваренный нетерминалом из левой части правила;
- `#`**`nonterm`** `->` `#`**`term`*** — правила, в правой части которых стоит цепочка терминалов.

Нужно заметить, что по практическим соображениям очень часто необходимо рассматривать не правила целиком, а лишь отдельные альтернативы. При этом имеются в виду альтернативы *верхнего уровня*, то есть те знаки «`|`», непосредственным родителем которых в дереве разбора определения грамматики будет само правило (символ «`->`»). Для решения этой проблемы мы вводим новые обозначения: знаком «`||`» мы обозначаем альтернативу верхнего уровня, причем при использовании этого знака в выражении для правила, порядок определения альтернатив в грамматике не учитывается. Так, если выражение «`N -> x | N x`» не допускает правила, в котором альтернативы идут в обратном порядке, то выражение «`N -> x || N x`» из допускает. Таким образом, знак «`|`» не коммутативен, а «`||`» — коммутативен.

В дополнение к «`||`» введем еще одно обозначение: кроме «`->`», можно использовать «`|>`», обозначающий, что имеется в виду не правило целиком, а какие-то его альтернативы верхнего уровня. Так, например, выражение «`N -> x || N x`» не допускает правило «`N -> x | N x | x y`», а то же выражение со знаком «`|>`» («`N |> x || N x`») — допускает.

Также нам понадобится оператор, которому соответствуют все альтернативы данного правила, кроме указанных. Для этой цели мы используем обозначение «{*}». Такое оператор не сделает язык запросов более сильным с точки зрения выбор множеств, но он поможет сопоставлять все неуказанные альтернативы некоторой переменной, которую впоследствии сможет использовать то преобразование, которое будет обрабатывать результаты запроса.

Осталось сказать, что множества, выбранные по различным запросам можно объединять, пересекать или вычитать. Для этого используются следующие обозначения:

- `[запрос] операция [запрос]`, где операция:
  - `|` - объединение,
  - `&` - пересечение,
  - `\` - вычитание множеств.

На этом мы остановимся, не затрагивая дальнейшие тонкости, связанные с языком запросов.

Использование запросов и преобразований

Всякое преобразование работает с результатами некоторого запроса, который указывается перед телом преобразования:

```
map [запрос] {
      // тело преобразования
}
```

Запрос определяет переменные, которые могут быть использованы в теле преобразования.

```
map [#nonterm |> #nonterm op:#term #nonterm] {
      rule BinOp |> op;
}
```

Это преобразование строит правило, в правой части которого перечислены все терминалы, стоящие между двумя нетерминалами в какой-либо альтернативе в исходной грамматике. Нетерминал в левой части этого правила получает имя BinOp. Тело преобразования выполняется для каждого значения переменной op, выбранного по запросу. Таким образом, если в исходной грамматике были, например, такие правила:

```
S -> S '+' M | M;
M -> M '*' F | F;
```

то в результате применения нашего преобразования получится правило

```
BinOp -> '+' | '*';
```

Здесь имеется важный момент: взаимодействие различных преобразований в нашей системе обеспечивается с помощью *глобального репозитория*. Задача состоит в том, чтобы каждое преобразование могло обращаться к объектам, созданным другими преобразованиями (или тем же преобразованием, но вызванным ранее с другими аргументами), причем желательно, чтобы не было необходимости связывать зависимые преобразования статически — во время компиляции. В частности, для нашего примера важно не создавать многочисленные копии нетерминала BinOp, определяемые по-разному, а, напротив, добавлять альтернативы к определению одно и того же нетерминала. То есть переменная BinOp должна связываться с одним и тем же объектом при каждом запуске преобразования.

Во многих системах для этих целей применяются журналы (traces [19]), хранящие информацию о том, какой объект является результатом какого преобразования, однако нам такой подход представляется не очень удобным. Мы делаем так: всякий создаваемый объект помещается в глобальный репозиторий и может быть извлечен оттуда по значениям *ключевых полей*. Понятие ключевого поля соответствует понятию первичного ключа или уникального индекса в СУБД. Например, если мы хотим получить ранее созданный символ с именем «BinOp» (или создать его, если он не был создан ранее), нам достаточно написать следующее:

```
BinOp[name="BinOp"]
```

Поиск осуществляется по двум параметрам: классу объекта (*символ*, определяется контекстом описания переменной) и значению атрибута name (все атрибуты для поиска и их значения указываются в квадратных скобках после имени переменной). Поскольку имя (атрибут name) — наиболее распространенный первичный ключ, мы предоставляем

сокращенный синтаксис для случая, когда поиск ведется по нему:

```
BinOp
```

Эта строка делает в точности то же, что и предыдущая, но имя переменной теперь само по себе является ключом для поиска.

С помощью этого простого механизма (в сочетании с метаданными, о которых будет сказано ниже) можно легко и естественно обеспечивать связь между различными преобразованиями, не связывая разработчика статическими зависимостями.

Метаданные

В большинстве существующих средств разработки grammarware входной язык не ограничивается простым определением грамматики, в него встраиваются различные расширения: приоритеты и ассоциативность операций, семантические акции или схемы трансляции в генераторах компиляторов, синтезируемые и наследуемые атрибуты, правила форматирования кода и так далее.

Фактически каждое средство разработки определяет собственный вид *метаданных*, которыми дополняется грамматика. Для каждого конкретного средства разработки метаданные имеют свой собственный вид и семантику: они встроены во входной язык и обрабатываются определенным фиксированным способом. Например, семантические акции в ANTLR пишутся на языке Java с небольшими расширениями, заключаются в фигурные скобки и помещаются в левых частях правил. Такая жесткая фиксация метаданных, конечно, является ограничением, но узкоспециализированным системам большая гибкость и не требуется.

Нашей системе не требуется расширять грамматику для какой-то конкретной цели, ведь все, что она умеет делать — предоставлять разработчику возможность обрабатывать грамматику так, как ему хочется. Другой вопрос, что разработчику может быть весьма удобно писать трансформации, если во входной грамматике, кроме собственно синтаксической структуры языка, будут содержаться еще какие-то данные. Так, например, разработчик, создающий трансформацию, которая по входной грамматике строит форматировщик кода (code formatter), мог бы предполагать, что синтаксические конструкции, играющие роль блоков, внутри которых нужно увеличивать отступ, каким-то образом отмечены. В принципе, для этой цели подойдет конвенция именования: имена соответствующих нетерминалов могут, скажем, заканчиваться на «Block», но полагаться на имена не очень хорошо, да и не всегда это получится. Вместо того, чтобы использовать сомнительный и недостаточно гибкий механизм именования, мы предлагаем использовать следующий подход: разрешим сопоставлять любому элементу грамматики набор метаданных, с которым априори не ассоциирована никакая семантика — как интерпретировать эти данные, решат те трансформации, которые будут применяться к такой грамматике.

Итак, каждому элементу можно приписать некий набор атрибутов, у каждого из которых есть имя и, возможно, значение. Да, значения может не быть, в таком случае приписывание атрибута эквивалентно атрибуту с булевским значением: если атрибут приписан, true, если нет — false. Поддерживаются следующие типы значений:

- целое число,
- строка,
- идентификатор (просто слово),
- набор атрибутов.

Атрибуты и их значения указываются в квадратных скобках после имени (или обозначения) символа:

```
'+':[grName="Plus", proirity=Low]
```

Здесь значение атрибута grName — строка «Plus», а значение атрибута priority — идентификатор Low.

Поскольку никакая семантика не приписывается атрибутам системой, никаких проверок относительно их значений не происходит. Так, например, атрибут grName может иметь, скажем, числовое значение, и система не будет возражать. С другой стороны, разработчик может создать преобразование, которое будет генерировать сообщения об ошибках, обнаруживая неправильные в значения атрибутов.

Для удобства мы введем макроопределения: группы атрибутов, в которым можно обращаться по имени и приписывать их различным объектам многократно. Описываются группы следующим образом:

```
GROUP_NAME = [attr1=value1, attr2=value2, ...];
```

А использовать их можно так:

```
symbol:GROUP_NAME
```

Метаданные могут использоваться в запросах для уточнения критериев поиска. Кроме того, преобразования, строящие на выходе грамматику, могут добавлять метаданные к ее элементам:

```
rule Op[name=Plus*, priority:=Low] |> '+';
```

Заметим, что в приведенном примере атрибут name используется для поиска (к нему применяется знак «=»), а атрибут priority назначается (используется знак «:=»).

Теперь, когда описаны все основные концепции, составляющие наш подход, мы готовы продемонстрировать его в действии.

Использование предложенного подхода

Мы постараемся продемонстрировать преимущества описанного подхода на довольно простом примере: анализаторе арифметических выражений.

Язык выражений задается следующей грамматикой:

```
Expression -> num | '('Expression')' | Expression Operation
Expression;
Operation -> '+' | '*' | '-' | '/' | '^';
```

Здесь num — терминал, обозначающий числовой литерал (мы опускаем лексические определения для краткости). Как видно, эта грамматика неоднозначна: не учитываются приоритеты и ассоциативность операций.

Вместо того, чтобы вручную устранить неоднозначность, напишем преобразование, которое сделает это. В конечном итоге наше преобразование будет работать для любой грамматики, то есть будет повторно используемым модулем.

Поскольку из исходной грамматики приоритеты и ассоциативность операций извлечь невозможно, мы воспользуемся метаданными, чтобы передать эту информацию нашем преобразованию. Сделаем три макроопределения:

```
PLUS = [grName = 'Plus'; assoc=left; arg='Plus'];
MULT = [grName = 'Mult'; assoc=left; super='Plus'; arg='Pow'];
POW = [grName = 'Pow'; assoc=right; super='Mult'];
```

Здесь атрибут grName фиксирует название группы операций, assoc — ассоциативность, super — группу операций, с приоритетом меньшим на единицу, arg — группу операций с приоритетом большим на единицу. Так, например, MULT характеризует операции, аналогичные умножению: левоассоциативные, стоящие по приоритету между сложениям и возведением в степень.

Преобразуем нашу грамматику так, чтобы сопоставить знакам операций соответствующие группы атрибутов:

```
map [Expression -> def:{*}] {
      rule Expression -> def;
}
map [Operation -> {*}] {
      rule Operation -> '+':PLUS | '-':PLUS
                      | '*':MULT | '/':MULT
                      | '^':POW ;
}
```

Теперь напишем преобразование, строящее по исходной грамматике с такими метаданными однозначную грамматику. Наши операции должны выстроиться по возрастанию приоритета, причем в самом конце этой «цепочки» будут уже не бинарные операции, а числа и выражения в скобках. Для начала выделим для этих конечных элементов отдельное правило:

```
map [Expression -> atoms:{*} || Expression Op Expression] {
      rule Atom -> atoms;
}
```

Вот результат применения этого преобразования:

```
Atom -> num | '('Expression')';
```

Теперь сгруппируем знаки операций по приоритетам (то есть по значениям атрибута grName):

```
map [Operation |> op:#symbol] {
      rule Op[op.grName + 'Op'] |> op;
}
```

Выражение Op[op.grName + 'Op'] создает новый нетерминал с указанным именем (например, «PlusOp» для символов, у которых grName="Plus") или находит ранее созданный. Таким образом мы получим следующий результат:

```
PlusOp -> '+' | '-' ;
MultOp -> '*' | '/' ;
PowOp -> '^' ;
```

Теперь обратимся к общему виду правил для операций, которые должны у нас получиться:

```
S -> A | A Op S;                                                    (1)
```

для правоассоциативных операций (Op — знак операции, A — аргумент, то есть операции высшего приоритета), а для левоассоциативных — так:

```
S -> A | S Op A;                                                    (2)
```

Заметим, что первая альтернатива в обоих случая одинакова, причем именно по этой альтернативе операции выстраиваются в цепочку:

```
Expression -> Plus -> Mult -> Pow -> Atom
```

Напишем преобразование, которое «строит эту цепочку»:

```
map [Operation |> op:#symbol] {
      rule Super[op.super || 'Expression'] -> Operation[op.grName];
}
```

Входными данными являются все символы, стоящие в правой части определения нетерминала Operation, то есть все знаки операций. В этом преобразовании использовано ранее не упоминавшееся обозначение:

```
Super[op.super || 'Expression']
```

Здесь знак «||» обозначает «ленивое или» в следующем смысле: если значение op.super определено, то выражение «op.super || 'Expression'» имеет значение op.super, а если нет — то значение 'Expression'. Таким образом, для операций «Pow» будет построено правило

```
Mult -> Pow ;
```

поскольку в группе атрибутов POW есть атрибут super и его значение — 'Mult', а для «Plus» будет построено правило

```
Expression -> Plus ;
```

поскольку в соответствующей группе нет атрибута super. Проще говоря, с помощью операции «||» мы задали *значение по умолчанию* для атрибута super.

Осталось создать вторые альтернативы в правилах вида (1) и (2). Они различаются в зависимости от ассоциативности операции:

```
map [op:#symbol[assoc=left]] {
      rule Operation[op.grName] -> Arg[op.arg || 'Factor']
                                 | Operation Op[op.grName + 'Op'] Arg;
}
```

```
map [op:#symbol[assoc=right]] {
      rule Operation[op.grName] -> Arg[op.arg || 'Factor']
                                 | Arg Op[op.grName + 'Op'] Operation;
}
```

Параметры запроса позволяют различать значения атрибута assoc — для каждого из них мы строим свой тип альтернативы. Этих преобразования создают по две альтернативы в каждом правиле. Это в некотором роде дублирует действия предыдущего преобразования, однако позволяет сделать код более читаемым и назначить значение по умолчанию для атрибута arg. Дублирующиеся альтернативы автоматически отбрасываются.

В конечном итоге мы получим однозначную грамматику:

```
Expression -> Plus ;
PlusOp -> '+' | '-' ;
Plus -> Mult | Plus PlusOp Mult ;
MultOp -> '*' | '/' ;
Mult -> Pow | Mult MultOp Pow ;
PowOp -> '^' ;
Pow -> Atom | Atom PowOp Pow ;
Atom -> num | '('Expression')' ;
```

Хотелось бы еще избавиться от левой рекурсии. Можно было сделать это сразу, изменив наше преобразование для левоассоциативных операций (#empty обозначает ε-продукцию):

```
map [op:#symbol[assoc=left]] {
      rule Rest[op.grName + 'Rest']
            -> Op[op.grName + 'Op'] Arg[op.arg] Rest
            | #empty ;
      rule Operation[op.grName] -> Arg Rest;
}
```

Однако лучше создать повторно используемое преобразование, не завязанное на структуру исходной грамматики и наши метаданные:

```
map [N:#nonterm -> N a:* || b:{*}] {
      rule R[N.grName + 'Rest'] -> a R | #empty;
      rule N -> b R;
}
```

В итоге мы получим однозначную грамматику без левой рекурсии. Заметим сразу, что добавление к этой грамматике новых операций — очень простая задача: достаточно лишь дописать их в исходную грамматику и сопоставить правильные метаданные, все остальное (создание не такого уж малого количества правил) произойдет автоматически за счет универсальности созданных преобразований.

Теперь обратимся к другим видам преобразований. Для начала извлечем из нашей грамматики классы для вершин синтаксического дерева. Базовые классы «Вершина», «Бинарная операция» и «Число» мы опишем вручную, а всех наследников создадим автоматически. Вот базовые классы (описание дается на языке Emfatic [21]):

```
abstract class Expression {
}

abstract class BinaryExpr extends Expression {
      val left : Expression;
      val right : Expression;
}

class Num extends Expression {
      attr value : int;
}
```

Теперь напишем преобразование, строящее подклассы класса BinaryExpr:

```
map [Operation -> op:#term] {
      new EClass {
            name = op.grName;
            super += BinaryExpr;
      }
}
```

Это все, что нужно для создания всех подклассов BinaryExpr (в нашем примере их было бы пять — по одному на каждую операцию), вот один из таких классов:

```
class Plus extends BinaryExpr {
}
```

Теперь осталось построить входное описание для какого-нибудь генератора синтаксических анализаторов. Для примера возьмем ANTLR. Нам нужно построить семантические акции, создающие объекты наших классов. Напомним, для генерации текста мы используем StringTemplate — все, что находится внутри двойных угловых скобок — шаблон для генерации текста. Вот первое преобразование:

```
map [* |> sym:#nonterm] <<
      return <sym.name>;
>>
```

Для правил, в правой части которых стоит ровно один нетерминал, достаточно вернуть его. Для правил, в правой части которых стоит ровно один терминал (это операции), вернем новый объект:

```
map [Operation |> op:#term] <<
      return new <op.grName>();
>>
```

Когда операция встречается в инфиксной форме, запишем данные в ее атрибуты:

```
map [* |> left:#symbol op:#symbol right:#symbol] <<
      BinaryExpr result = <op.name>;
      op.setLeft(<left.name>);
      op.setRight(<right.name>);
      return result;
>>
```

Приведенный пример демонстрирует исключительную расширяемость, которая достигается за счет применения предлагаемого подхода. Изменения грамматики (в разумных пределах) автоматически учитываются преобразованиями, что делает разработку гораздо проще. Система перестала быть монолитной: вся информация распределена по преобразованиям, причем ее можно повторно использовать.

Это происходит благодаря гибкости модельно-ориентированной разработки и разделению различных аспектов системы. На самом деле концепция преобразований сродни идеям, применяемым в аспектно-ориентированном программировании [12]: запросы аналогичны точкам встраивания (pointcuts), а преобразования — встраиваемому коду (advice). Аспектно-ориентированное программирование возникло в связи с необходимостью разделять программные системы на модули еще более гибко, чем это позволяет ООП, так что не удивительно, что эти идеи пригодились и нам.

Заключение

Данная работа является представлением нового подхода к разработке grammarware, основанного на MDD и использующего принципы аспектно-ориентированного программирования для разделения различных аспектов системы.

В работе показано как предлагаемый подход может изменить процесс разработки ПО, связанного с грамматиками: система становится более модульной, ее легче модифицировать и расширять. Сам процесс разработки сохраняется для повторного использования в виде преобразований.

Наш подход основывается на новых инструментальных средствах (предметно-

ориентированных языках, описанных в данной работе), а также на изменении самого понятия разработка: артефакты не модифицируются вручную, а генерируются с помощью указанных инструментов.

Описанные принципы предстоит развивать и уточнять прежде, чем их смогут использовать разработчики промышленных систем, но уже сейчас понятно, что большинство проблем, описанных в работе [3] могут быть решены в рамках предлагаемого подхода.

Литература

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ — М.: Бином, 2001. — 560 с.
2. Мейер Б. Объектно-ориентированное конструирование программных систем / Пер. с англ. — М.: Русская редакция, 2005. — 1232 с.
3. Klint P., Lammel R., Verhoef C. Toward an engineering discipline for GRAMMARWARE — ACM Transactions on Software Engineering Methodology, Vol. 14:3, 2005. — стр. 331 - 380
4. Atkinson C., Kuhne T. Model-driven development: a metamodeling foundation — IEEE Software, Vol. 20:5, 2003. — стр. 36 - 41.
5. Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai, G. Composing domain-specific design environments — Computer, Vol. 34:11, 2001 — стр. 44 - 51.
6. Parr T. The definitive ANTLR reference — The Pragmatic Bookshelf, 2007.
7. Gagnon E., Hedren L. SableCC, an Object-Oriented Compiler Framework — Technology of Object-Oriented Languages and Systems, 1998.
8. Efftinge S. oAW xText: A framework for textual DSLs — Eclipse Summit 2006 Workshop: Modeling Symposium, 2006.
9. Jouault F., Bezivin J., Kurtev I. TCS: a DSL for the specification of textual concrete syntaxes in model engineering — Proceedings of the 5th international conference on Generative programming and component engineering, 2006.
10. Klint P. A Meta-Environment for generating programming environments — ACM Transactions on Software Engineering and Methodology, Vol. 2:2, 1993. — стр. 176 - 201.
11. Clavel M., Duran F., Eker S., Lincoln P., Marti-Oliet N., Meseguer J., TalcottC. The Maude 2.0 System — Rewriting Techniques and Applications in Lecture Notes in Computer Science, Vol. 2706, 2003 — стр. 76 - 87
12. Kiczales G., Lamping J., Lopes C., Hugunin J., Hilsdale E., Boyapati C. Aspect-oriented programming — Xerox Corporation, 2002.
13. Чарнецки К., Айзенекер У. Порождающее программиование: методы, инструменты, прменение. Для профессионалов. — СПб.: Питер, 2005. — 731 с.
14. Extended Backus-Naur Form / ISO/IEC 14977 : 1996(E) — ISO, 1996
15. Фаулер М. UML. Основы — М.: Символ-Плюс, 2006. — 192 стр.
16. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии, инструменты — М.: Вильямс, 2003 — 768 с.
17. Clark T., Sammut P., Willans J. Applied metamodeling: a foundation for language driven development — Ceteva, 2008.
18. Steinberg D., Budinsky F., Paternostro M., Merks E. EMF: Eclipse Modeling Framework, Second Edition — Addison Wesley, 2007.
19. Jouault F., Kurtev I. On the architectural alignment of ATL and QVT — Proceedings of the 2006 ACM symposium on Applied computing, 2006.
20. Parr T. Enforcing strict model-view separation in template engines — Proceedings of the 13th international conference on World Wide Web, 2004.

21. Daly C. Emfatic Language for EMF Development — IBM alphaWorks, 2004. // http://www.alphaworks.ibm.com/tech/emfatic