Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ

Дипломная работа студента 544 группы
Чемоданова Ильи Сергеевича

Научный руководитель	.....	Я.А. Кириленко
ст. преподаватель	/ подпись /	
Рецензент	.....	А.С. Лукичев
к.ф.-м.н., ассистент	/ подпись /	
“Допустить к защите”		
заведующий кафедрой,	.....	А.Н. Терехов
д.ф.-м.н., профессор	/ подпись /	

Санкт-Петербург
2007

Содержание

1 Введение

Задачи автоматизированного реинжиниринга программ [1] выдвигают особые требования [14] к генераторам синтаксических анализаторов. Во многом это связано с тем, что теория синтаксически управляемой трансляции развивалась одновременно с появлением языков, сейчас называемых устаревшими (legacy languages). Такие языки проектировались, когда еще не были получены основные теоретические результаты, положенные в основу наиболее распространенных современных генераторов. Поэтому устаревшие языки трудны для синтаксического анализа даже с использованием современных инструментов, которые значительно упрощают создание анализаторов.

В статье [14] перечислены генераторы, опыт применения которых оказался неудачным при решении задач реинжиниринга. С другой стороны, в проекте по разработке системы *Modernization Workbench* [20] (в котором автор участвует в течение двух с лишнем лет) имеется опыт использования инструментов YACC++, ANTLR и Bison, который также нельзя назвать удачным.

В документации для описания грамматик устаревших языков используется не форма Бэкуса-Наура (БНФ [4]), а синтаксические диаграммы (syntax diagrams), которые обладают значительно большими возможностями. Например, в них часто указывают число повторений однотипных элементов при различных перечислениях, что трудно выразить с помощью БНФ. Также часто встречаются перечисления (например, в языке *Cobol*), где каждый элемент встречается ровно один раз. Синтаксис таких конструкций удобно задавать с помощью перестановки [16]. Среди современных инструментов конструкция перестановки есть только у ASF+SDF [25].

Для устаревшего языка сложно (а зачастую и невозможно) задать однозначную контекстно-свободную грамматику. Необходимо существенно преобразовать его спецификацию, которая приводится в документации, чтобы получить такую грамматику, но при этом она перестает быть сопровождаемой [14]. Поэтому устаревший язык обычно задается с помощью неоднозначной контекстно-свободной (или даже контекстно-зависимой) грамматики, в то время как инструменты порождают анализаторы лишь для некоторого подмножества однозначных контекстно-свободных грамматик ($LR(k)$ или $LL(k)$ с фиксированным k).

При поддержке нескольких диалектов языка необходима возможность повторного использования элементов грамматики для упрощения кода и повышения сопровождаемости. В то же время многие языки имеют синтаксически похожие конструкции, поэтому можно переиспользовать некоторые части грамматики одного языка при создании анализатора другого языка. Следовательно генератор должен обладать средствами повторного использования элементов грамматик.

В результате обзора современных инструментов было выявлено, что ни один из них полностью не отвечает этим требованиям. Современные генераторы предоставляют не достаточно гибкие и удобные средства как для задания грамматик устаревших языков, так и для работы с неоднозначными грамматиками.

Задача данной работы — предложить прототип инструмента для создания синтаксических анализаторов с выразительным и удобным входным языком (см. часть 2.1), в большей степени отвечающего требованиям автоматизированного реинжиниринга программ по сравнению с существующими разработками.

Структура дипломной работы Требования к генератору анализаторов, связанные со спецификой устаревших языков, подробно описаны в части 2.1. Краткое описа-

ние инструментов автоматизации создания анализаторов, рассмотренных при обзоре, приводится в части 2.2. Результаты обзора сформулированы в части 2.3. В части 3 описаны возможности предлагаемого инструмента (часть 3.1) и его реализация (части 3.2 и 3.3). Полная спецификация входного языка генератора *YARD* приводится в приложении A. Чтобы показать выразительность предлагаемого входного языка, в приложениях B и C задана одна и та же грамматика (для разбора входного файла инструмента *YARD*) в нотации *ocamlyacc* и *YARD*, соответственно. Результаты работы сформулированы в части 4.

2 Обзор существующих инструментов

В части 2.1 формулируются требования к инструменту, выявленные при изучении предметной области. Затем (часть 2.2) кратко описаны рассмотренные инструменты, и подводятся итоги обзора (часть 2.3).

2.1 Требования к инструменту

Одним из главных требований к инструменту при решении задач автоматизированного реинжиниринга программ является работа с неоднозначными грамматиками, с помощью которых очень часто описываются устаревшие языки.

Некоторые типы неоднозначностей (конфликтов) можно устранить путем соответствующих преобразований грамматики [9], но при этом она может значительно усложниться, что отрицательно сказывается на ее сопровождаемости. Поэтому входной язык генератора и внутренний алгоритм должны предоставлять возможности по эффективному разрешению неоднозначностей (типы и механизмы их разрешения подробно рассмотрены в статье [8]). Одним из способов разрешения неоднозначностей во время разбора является использование предикатов (резольверов [7]).

При поддержке нескольких диалектов языка (например, в системе *Modernization Workbench* [20] поддерживаются 18 диалектов языка *Cobol*, 4 диалекта для языка *Natural*, а также 2 диалекта языка *PL/I*) можно (и нужно в соответствии с принципами качественного кода [5]) переиспользовать общие части грамматик этих диалектов. Помимо этого, многие языки имеют синтаксически похожие конструкции, например, арифметические выражения, поэтому можно повторно использовать некоторые части грамматики одного языка при создании анализатора другого языка. То есть инструмент должен обладать средствами повторного использования элементов грамматик. Одним из основных средств повторного использования является модульность (разбиение спецификации разбираемого языка на несколько частей). Разделение грамматики на модули необходимо не только для повторного использования каких-то ее частей, но и для упрощения процесса разработки и сопровождения, поскольку таким образом повышается ее управляемость и снижается сложность.

Для уменьшения грамматики, а также ее упрощения, инструмент должен предоставлять высокоуровневые средства для ее задания. Одним из таких средств является параметризация одних правил другими. Такая параметризация нужна для введения общих концепций и их повторного использования. Проводя параллель с языками программирования, правила, параметризованные другими правилами, соответствуют шаблонам (*templates*) в *C++* и настраиваемым типам (*generics*) в *C#* и *Java*.

Чтобы обосновать необходимость и раскрыть возможности макроправил, рассмотрим небольшой пример. Во всех языках есть конструкции, синтаксис которых удобно определять с помощью списка (перечисления), то есть последовательности однотипных элементов, разделенных каким-то специальным символом — так называемая итерация с разделителем (в технологическом комплексе SYNTAX [6] с этой целью введена специальная операция #, которая называется *звездочкой Цейтина*). Например, в списке параметров функции или процедуры разделителем обычно является запятая, а операторы (statements) во многих языках пишут через точку с запятой. Правило для разбора таких списков удобно параметризовать правилом для разбора элемента списка и разделителем (или правилом для разделителей, если их

несколько):

```
list<<item delimiter>>
:  item ( delimiter item )*
;

func
:  typeName funcName list<<param ",">> funcBody
;

funcBody
:  "{" list<<stmt ";">> "}"
;
```

В этом примере мы "сэкономили" одно правило (без использования параметризации пришлось бы создать два списка — `paramList` и `stmtList`). Обычно в грамматике таких списков оказывается гораздо больше, и экономия оказывается более существенной, в то же время мы приводим только один пример, где удобно использовать параметризацию.

Грамматика из примера очень похожа на двухуровневую грамматику Вейнгаардена [19] (Adriaan van Wijngaarden). При этом правило `list` в терминах Вейнгаардена называется гиперправилом (*hyper-rule*), а правила `item` и `delimiter` — метаправилами (*meta-rule*). Данный формализм был использован при описании синтаксиса и семантики языка Algol68 [2].

В синтаксических диаграммах, которые используются для описания устаревших языков, часто встречаются перестановки [16], с помощью которых определяются списки опций или атрибутов, где каждый элемент встречается только один раз. Обозначим перестановку как "[| a, b, c, ... |]", тогда упрощенное описание данных в языке *Cobol*[1] можно задать так

```
dataEntry
:  levelNumber dataName [| pictureClause, occursClause, usageClause |]
;
```

Таким образом, грамматика становится более строгой, с меньшим числом допущений и конфликтов, поскольку такой список обычно записывают следующим образом (конструкция перестановки есть только в ASF+SDF):

```
dataEntry
:  levelNumber dataName ( pictureClause | occursClause | usageClause )+
;
```

Часто различные списки и другие последовательности однотипных элементов имеют фиксированную максимальную и/или минимальную длину. Например, в языке *Unisys Work Flow Language* (WFL [46]) путь к файлу или каталогу не может иметь более 19 уровней вложенности. Упрощенная грамматика для обработки имени файла в языке WFL может выглядеть так:

```
fileName
:  NODE_NAME ( "/" NODE_NAME )[0, 19]
;
```

[1] Здесь рассматривается диалект фирмы IBM [45].

В этом примере конструкция `( "/" NODE_NAME )[0,19]` означает повторение `("/" NODE_NAME)` от 0 до 19 раз включительно. К сожалению, ни один из современных инструментов не поддерживает такой возможности, поэтому обычно это же правило записывают следующим образом:

```
fileName : NODE_NAME ( "/" NODE_NAME )* ;
```

Из примера видно, что грамматика в таком случае содержит допущение и не до конца соответствует спецификации. Часто такая запись еще и вносит в грамматику дополнительные конфликты.

В настоящее время ни один из популярных инструментов не отвечает всем перечисленным выше требованиям. В то же время нельзя не отметить, что современные инструменты (см. таблицу 1) обладают такими удобными средствами для задания грамматик, как

- операции РБНФ[2] (обычно их обозначают через “*“, “+” и “?”);
- правила с параметрами, которые значительно упрощают трансляцию с учетом контекста (в терминах атрибутной грамматики [13] такие параметры называются наследуемыми атрибутами);
- специальные конструкции для разрешения неоднозначностей в грамматике — предикаты (резольверы [7]);
- именование семантических значений (вместо использования $1, $2, $3, ...), что уменьшает число возможных ошибок при обращении к ним.

Таким образом, генератор синтаксических анализаторов должен обладать как уже хорошо зарекомендовавшими себя возможностями, так и предоставлять дополнительные средства для задания грамматики и разрешения неоднозначностей в ней, описанные выше.

2.2 Рассмотренные инструменты

В статье [8] проводится подробный анализ возможностей следующих инструментов:

- ANTLR [24] (ANother Tool for Language Recognition) — один из самых популярных сейчас генераторов. Главным его создателем является Теренс Пар (Terence Parr) из университета Сан-Франциско (США). Использует $LL(k)$-алгоритм.
- ASF+SDF [25] (Algebraic Specification Formalism + Syntax Definition Formalism) — генератор с широкими возможностями, но достаточно сложным входным языком. Является $SGLR$-инструментом (Scannerless, Generalized-LR).
- Bison [26] — развитие инструмента YACC. Все грамматики, созданные для оригинального YACC, будут работать и в Bison. Является одним из самых популярных и совершенных “потомков” YACC. При включении соответствующей опции использует GLR-алгоритм (по умолчанию $LALR$).

[2] Расширенная форма Бэкуса-Наура [4]

- Coco/R [28] (COmpiler COmpiler generating Recursive descent parsers) — академический проект, разрабатываемый в университете Линц (University of Linz), в Австрии. Данный *LL*-инструмент используется в Rotor[3] (некоммерческой реализации платформы .NET) для создания компилятора и различных по назначению анализаторов C#.

- Elkhound [33] — позиционируется как быстрый и удобный *GLR*-инструмент, созданный в университете Беркли (США), тем не менее обладает достаточно "бедным" входным языком (например, он не поддерживает конструкций расширенной формы Бэкуса-Наура).

- JavaCC [35] (Java Compiler Compiler) — популярный *LL*-инструмент для создания синтаксических анализаторов на языке Java[4], главный конкурент ANTLR. Разрабатывается с 1996 года при поддержке Sun Microsystems.

- Menhir [38] — реализация YACC с целевым языком Objective Caml, но имеющий значительно большую функциональность даже в сравнении с GNU Bison. Его создатели (Francois Pottier и Yann Regis-Gianas) являются сотрудниками Французского национального института информатики и автоматизации (INRIA).

- SLK [42] (Strong LL(k)) — позиционируется как единственный настоящий $LL(k)$-генератор. Этот инструмент умеет преобразовывать грамматики (в нотации IEEE, ISO или YACC) в свой собственный формат, а также устранять из них левую рекурсию.

Помимо перечисленных выше, интерес представляет *GLR*-инструмент Dypgen [32], стабильная версия которого появилась только несколько месяцев назад (и в связи с этим не попавшего в обзор). Этот генератор обладает следующими особенностями: позволяет добавлять и удалять правила разбора прямо во время синтаксического анализа, возвращает список результатов разбора (если возможен только один способ разбора, список состоит из одного элемента), также предоставляет интересную нотацию для задания приоритетов.

Существует большое число генераторов (AnaGram [23], ClearParse [27], CompTools [29], CppCC [30], CUP [31], Grammatica [34], Lemon [36], LLGen [37], PRECC [39], RDP [40], Rie [41], Styx [43], TPG [44] и другие), которые не были упомянуты в статье, поскольку не предоставляют каких-то уникальных возможностей, особой функциональности, позволяющей упростить разработку синтаксических анализаторов, а иногда даже не обладают некоторыми возможностями, ставшими стандартом де-факто для современных инструментов (см. таблицу 1). В рамках данной работы эти генераторы были рассмотрены для полноты обзора.

2.3 Результаты обзора

Резюмируя статью [8], можно отметить, что в настоящее время нет инструмента для создания анализаторов, который бы полностью отвечал требованиям автоматизированного реинжиниринга (см. часть 2.1).

Результаты сравнения всех рассмотренных инструментов по нескольким из основных показателей можно найти в таблице 1.

[3] http://research.microsoft.com/programs/europe/rotor/

[4] http://java.sun.com/

Название инструмента	Тип алгоритма	Возможности входного языка				
		РБНФ	Макро-правила	Наследуемые атрибуты	Предикаты	Именование сем. значений
AnaGram	$LALR(1)$	+	−	−	−	+
ANTLR	$LL(k)$	+	−	+	+	+
ASF+SDF	GLR	+	−	−	−	+
Bison	$LALR(1)$	−	−	−	−	−
ClearParse	$LL(1)$	+	−	−	−	−
Coco/R	$LL(1)$	+	−	+	+	+
CompTools	$LL(1)$	+	−	−	−	+
CppCC	$LL(k)$	+	−	+	−	+
CUP	$LALR(1)$	−	−	−	−	+
Dypgen	GLR	−	−	−	−	+
Elkhound	GLR	−	−	−	−	+
Grammatica	$LL(k)$	+	−	−	−	−
JavaCC	$LL(k)$	+	−	+	+	+
Lemon	$LALR(1)$	−	−	−	−	+
LLGen	$LL(1)$	+	−	−	−	+
Menhir	$LR(1)$	+	+	−	−	+
PRECC	$LL(k)$	+	−	+	−	+
RDP	$LL(1)$	+	−	+	−	+
Rie	$LALR(1)$	−	−	+	−	−
SLK	$LL(k)$	+	−	−	−	−
Styx	$LALR(1)$	−	−	−	−	+
Toy (TPG)	$LL(1)$	+	−	+	−	+

Таблица 1: Тип внутреннего алгоритма и основные возможности входных языков различных генераторов.

На практике, при анализе устаревших языков наиболее удобны инструменты [14], использующие GLR-алгоритм [18]. В то же время, при анализе современных языков очень популярны LL-инструменты с резольверами.

Среди LL-генераторов предпочтительней других выглядят ANTLR, JavaCC и Coco/R. Все три инструмента порождают анализаторы, реализованные по методу рекурсивного спуска, поэтому сгенерированный код легко прочесть и отладить, в отличие от кода, порождаемого LR-генераторами. В этой троице лидером по своим возможностям является ANTLR.

Среди GLR-инструментов (Elkhound , ASF+SDF, Bison, Dypgen) можно выделить Elkhound и Dypgen, поскольку ASF+SDF имеет сложный входной язык, а в Bison GLR-алгоритм реализован неэффективно.

Лучшие средства повторного использования предоставляет ASF+SDF, где модульность грамматики реализована на уровне спецификации входного языка. Неплохими средствами обладают Elkhound и Menhir, где возможно хотя бы разделить грамматику на достаточное число файлов, в отличие от ANTLR (где можно разбить грамматику только на два файла), JavaCC и Coco/R. В двух последних инструментах вся спецификация грамматики располагается в одном файле.

Из существующих генераторов только Menhir поддерживает параметризацию од-

них правил другими (макроправила), хотя, в отличие от ANTLR, Coco/R и JavaCC, не позволяет использовать наследуемые атрибуты и предикаты.

3 Реализация

В качестве удобной платформы и среды для создания инструмента был взят генератор анализаторов *YARD*[5], разрабатываемый Я.А. Кириленко. Далее создаваемый инструмент для краткости часто будет называться *YARD*. Языком реализации является язык OCaml [22].

3.1 Входной язык генератора

За основу для метаязыка генератора был взят язык, используемый в синтаксическом расширении, которое входит в поставку библиотеки *Ostap* [21]. Эта библиотека монадических комбинаторов [10, 11] разработана Я.А. Кириленко и Д.Ю. Булычевым. Основными достоинствами упомянутого языка являются:

- конструкции расширенной формы Бэкуса-Наура;
- параметризация одних правил другими;
- предикаты (резольверы);
- использование наследуемых атрибутов;
- применение OCaml-образцов (patterns) при именовании семантических значений (семантик), возвращаемых правилами;
- неявное описание нетерминалов (анонимные нетерминалы). Например, правило "`a : ( b | c ) d ( e | f ) ;`" содержит два анонимных нетерминала "`( b | c )`" и "`( e | f )`". Далее будем называть их группировкой альтернатив.

Этот язык выразителен и удобен, но анализатор, построенный с использованием комбинаторов, имеет в худшем случае экспоненциальную сложность разбора от длины входного потока. Это связано с тем, что такой анализатор перебирает все возможные варианты разбора входного потока и возвращает список благоприятных исходов. Наложив на язык, соответствующий семантике комбинаторов, определенное ограничение (которое тем не менее незначительно снижает его выразительность и удобство при практическом применении), возможно создать генератор анализаторов, использующий достаточно эффективный алгоритм (о котором подробнее говорится в части 3.2).

Это ограничение состоит в следующем: аргументы параметризованного правила не могут выступать в качестве синтезируемых или наследуемых атрибутов. Правило, параметризованное другими правилами, после подстановки всех фактических аргументов должно принимать обычный вид, то есть не допускается частичное задание параметров.

Далее правило, параметризованное другими правилами, будем называть для краткости параметризованным правилом или макроправилом. Правило, не являющееся макроправилом, будем называть обычным правилом (иногда просто правилом, если контекст это позволяет).

Отметим, что аргументами макроправила могут выступать терминальные символы (лексемы), нетерминальные символы (имена обычных правил) и макроправила

[5] *YARD* — Yet Another Recursive Descendent parser generator. © Я.А. Кириленко, 2004-2007

(но без аргументов). Аргументы макроправила, которые сами являются макроправилами, называются параметрами высшего порядка (*higher-order parameters*).

Помимо основных средств языка *Ostap* метаязык генератора поддерживает ряд конструкций, необходимость которых была обоснована в части 2.1, это

- расширенные регулярные выражения ("`( a b c )[1,5]`");
- перестановка ("`[| a, b, c |]`").

Полная спецификация метаязыка генератора приводится в приложении А.

3.1.1 Пример грамматики калькулятора

В качестве примера рассмотрим грамматику синтаксического анализатора программы-калькулятора в нотации *YARD* и сравним с аналогичной в обозначениях ANTLR.

На входе анализатор (созданный по рассматриваемой грамматике) принимает арифметическое выражение, а выдает результат его вычисления. Выражение может содержать скобки, а также следующие действия: сложение, вычитание, умножение, деление и возведение в степень.

В нотации *YARD* грамматика такого калькулятора имеет вид:

```
binExpr<<operand binOp>>
: l=operand r=( op=binOp r=operand { (op, r) } )*
  { List.fold_left ( fun l (op,r) -> op l r ) l r }
;

termOp: PLUS { ( +. ) } | MINUS { ( -. ) } ;

factorOp : MULT { ( *. ) } | DIV { ( /. ) } ;

powOp: "^" { ( ** ) } ;

powExpr: n=NUMBER { float n } | "(" e=expr ")" { e } ;

factor: res=binExpr<<powExpr powOp>> { res } ;

term: res=binExpr<<factor factorOp>> { res } ;

expr: res=binExpr<<term termOp>> { res } ;
```

Чтобы избежать дублирования кода, мы создали макроправило `binExpr` для бинарного выражения, которое параметризуется операндом и бинарной операцией. Начальным нетерминалом для разбора выражения является символ `expr`.

Для связывания семантического значения (возвращаемого правилом) с некоторым именем используются равенства вида `myBinding = myRule`. После успешного применения правила `myRule` к входному потоку, связанное имя `myBinding` будет содержать семантическое значение, возвращенное этим правилом. Связывание имеет ограниченную область видимости, которая начинается сразу после

его объявления, а заканчивается вместе с контекстом, внутри которого находится это связывание. Правая часть каждого правила является отдельным контекстом, внутри которого могут появляться вложенные контексты, задаваемые круглыми скобками. Например, в правой части макроправила `binExpr` выражение `( op=binOp r=operand { (op, r) } )*` определяет вложенный контекст, поэтому связывания `op` и `r` не будут видны (доступны) в контексте всего правила.

В обозначениях ANTLR аналогичная грамматика выглядит следующим образом:

```
options {
  language="Cpp";
}

{ #include <math.h> }

class CalcParser extends Parser;

expr returns [float res]
{ res = 0;
  float r = 0;
  bool plus = false;
}
: res=term
  ( ( PLUS  { plus = true;  } | MINUS { plus = false; } ) r=term
    { if (plus) res += r; else res -= r; }
  )*
;

term returns [float res]
{ res = 0;
  float r = 0;
  bool mult = false;
}
: res=factor
  ( ( MULT { mult = true; } | DIV { mult = false; } ) r=factor
    { if (mult) res *= r; else res /= r; }
  )*
;

factor returns [float res]
{ res = 0;
  float r = 0;
}
: res=powExpr ( POW r=powExpr { res = pow(res, r); } )* ;

powExpr returns [float res]
{ res = 0; }
: i:INT { res = atof(i->getText().c_str()); } | LPAREN res=expr RPAREN ;
```

Как видно грамматика в нотации ANTLR в два с лишнем раза длиннее (39 строк

против 18) грамматики для *YARD*. Хотя в ANTLR можно получить и более короткую грамматику, если использовать абстрактное синтаксическое дерево:

```
options {
  language="Cpp";
}

class CalcParser extends Parser;

options {
  buildAST = true; // uses CommonAST by default
}

expr: term ( ( PLUS^ | MINUS^ ) term )* ;

term: factor ( ( MULT^ | DIV^ ) factor )* ;

factor: powExpr ( POW^ powExpr )* ;

powExpr: INT | LPAREN! expr RPAREN! ;

{ #include <math.h> }

class CalcTreeWalker extends TreeParser;

expr returns [float r]
{ float a, b;
  r = 0;
}
: #(PLUS  a=expr b=expr) { r = a + b; }
| #(MINUS a=expr b=expr) { r = a - b; }
| #(MULT  a=expr b=expr) { r = a * b; }
| #(DIV   a=expr b=expr) { r = a / b; }
| #(POW   a=expr b=expr) { r = pow(a, b); }
| i:INT { r = atof(i->getText().c_str()); }
;
```

Здесь для обработки выражения используются два прохода. При первом проходе строится абстрактное синтаксическое дерево разбора (в автоматическом режиме), а затем при его обходе вычисляется значение выражения. По сравнению с первым вариантом грамматика стала не только на 6 строк короче, но и проще, удобнее для понимания и сопровождения.

Грамматика калькулятора в нотации *YARD* занимает меньше строк (18 против 33-х) и в сравнении со вторым вариантом грамматики в обозначениях ANTLR. Такой выигрыш получен, главным образом, за счет использования макроправил.

Пример, более содержательный, чем рассмотренный выше, можно найти в приложении C, где приводится грамматика в нотации *YARD* для анализа входного файла инструмента *YARD*. Анализатор, построенный на основе этой грамматики, в качестве результата своей работы возвращает дерево разбора. Чтобы показать вырази-

тельность входного языка инструмента *YARD*, в приложении В описана та же грамматика, но в нотации *ocamlyacc*.

3.2 Внутренний алгоритм

На практике, при анализе устаревших языков наиболее удобны инструменты [14], использующие GLR-алгоритм. В статье [14] подробно описано, почему задачи реинжиниринга трудно решать с помощью $LR(k)$ и $LL(k)$-инструментов.

Главным достоинством GLR-алгоритма является обработка неоднозначных грамматик. Анализатор, построенный с помощью данного алгоритма, в результате разбора строит не единственное дерево, а несколько деревьев — лес, который можно сократить, используя специальные фильтры. Стоит отметить, что по производительности такой анализатор, являясь некоторой "надстройкой" над LR-анализатором, незначительно ему уступает. На сегодняшний день в соотношении производительность / класс разбираемых языков GLR-алгоритм выглядит наиболее предпочтительно [15]. В то же время отладка исходного кода GLR-анализатора остается такой же трудной, как и у LR-анализатора.

Возможность GLR-анализатора возвращать список всех результатов разбора позволяет реализовать предикаты, как фильтры леса, полученного в результате синтаксического анализа (более подробно реализация предикатов описана в части 3.3.5).

В силу перечисленных выше причин в качестве внутреннего алгоритма для разрабатываемого инструмента было решено взять алгоритм GLR. Также было принято следующее техническое решение: не реализовывать самостоятельно данный алгоритм, а воспользоваться его существующей реализацией в генераторе Elkhound . Схема его использования такова: по грамматике для инструмента *YARD* строится эквивалентная грамматика для генератора Elkhound , после чего она обрабатывается инструментом Elkhound , который порождает GLR-анализатор на языке OCaml .

3.3 Генерация грамматики в нотации Elkhound

Elkhound не поддерживает конструкций расширенной формы Бэкуса-Наура, параметризации правил, предикатов и наследуемых атрибутов, а также не позволяет группировать альтернативы с помощью скобок, в отличие от *YARD*. Поэтому при генерации грамматики в нотации Elkhound из грамматики в обозначениях *YARD*, возникает задача предварительного преобразования грамматики к надлежащему (подходящему) виду.

3.3.1 Преобразование конструкций расширенной формы Бэкуса-Наура

Под расширенной формой Бэкуса-Наура понимаются следующие конструкции:

- `a*` — применение при разборе правила `a` нуль и более раз;
- `a+` — применение при разборе правила `a` один и более раз. Эта конструкция может быть выражена через предыдущую следующим образом: `a a*`;
- `a?` — опциональное (возможное) однократное использование при разборе правила `a`.

Существует несколько способов раскрытия перечисленных выше конструкций. Наиболее очевидным кажется такой: каждое их использование заменять вызовом нового правила (и при этом создавать это новое правило). Рассмотрим данный алгоритм на примере `a*`. Пусть есть грамматика (в нотации *YARD*), содержащая следующее правило:

```
stmtList : stmt ( SEMI stmt )* ;
```

В этом случае создается новое правило (в обозначениях Elkhound) вида

```
nonterm manyStmt {
  -> empty;
  -> SEMI stmt manyStmt;
}
```

где `empty` — обозначение пустой альтернативы, принятое в Elkhound . А правило `stmtList` переписывается следующим образом:

```
nonterm stmtList {
  -> stmt manyStmt;
}
```

Конструкции расширенной формы Бэкуса-Наура можно раскрывать и с помощью макроправил. Для рассмотренного выше примера с правилом `stmtList` макроправило выглядит так

```
many<<delim item>>
: (* empty *)
| delim item many<<delim item>>
;
```

При этом само правило `stmtList` принимает вид:

```
stmtList : stmt many<<SEMI stmt>> ;
```

Поскольку *YARD* поддерживает такую параметризацию, было принято решение при генерации грамматики для Elkhound раскрывать конструкции расширенной формы Бэкуса-Наура именно таким способом. В то же время эти преобразования имеют промежуточный характер, поскольку Elkhound не позволяет параметризовать одни правила другими, поэтому приходится выполнять дополнительные модификации грамматики, описанные в части 3.3.2.

В рассмотренном примере для конструкции `a*` правило `many` имеет лишь два параметра, хотя их число может быть и большим, а также они могут иметь более сложную структуру, например:

```
stmtList : stmt ( ( SEMI | NEWLINE ) stmt )* ;
```

Поэтому считается, что конструкции расширенной формы Бэкуса-Наура соответствует правило, имеющее только один параметр. Для конструкции `a*` это

```
many<<item>>
: (* empty *)
| item many<<item>>
;
```

В случае же когда требуется параметризация `many` несколькими правилами, создается новое дополнительное правило. Так для последнего примера будет сгенерирован код (в нотации Elkhound) вида:

```
nonterm yard_alt_1 {
  -> SEMI;
  -> NEWLINE;
}

nonterm yard_item_1 {
  -> yard_alt_1 stmt;
}

nonterm yard_many_1 {
  -> empty;
  -> yard_item_1 yard_many_1;
}

nonterm stmtList {
  -> stmt yard_many_1;
}
```

Поскольку Elkhound не позволяет группировать альтернативы, в этом примере помимо правила `yard_item_1` было еще создано и правило `yard_alt_1`. О преобразовании альтернатив, сгруппированных с помощью скобок, подробнее рассказывается в части 3.3.3. Можно отметить, что последний пример еще раз показывает насколько выразительна и удобна *YARD*-нотация по сравнению с Elkhound .

3.3.2 Преобразование макроправил

Как уже отмечалось, Elkhound не поддерживает правил, параметризованных другими правилами (среди современных инструментов такой функциональностью обладает только Menhir), поэтому каждое их использование приходится заменять вызовом нового, параллельно создаваемого правила, или вызовом уже сгенерированного правила.

Теперь рассмотрим алгоритм раскрытия макроправил более подробно. Обозначим через R множество всех правил грамматики до применения алгоритма, через R' — множество всех правил грамматики после применения алгоритма, а через M — множество макроправил, которые доступны в текущем контексте. Будем считать, что множество R — список, это необходимо только для определения текущего контекста. В этом списке правила находятся в порядке их расположения в файле с грамматикой. В текущем контексте, то есть в правой части текущего правила (не являющегося макроправилом), доступны только те макроправила, которые были объявлены выше. Перед началом алгоритма ни одно макроправило не определено ($M = \emptyset$) и нет обработанных правил ($R' = \emptyset$). Опишем алгоритм раскрытия макроправил по шагам:

1. Если $R \neq \emptyset$, то перейти к шагу 2, иначе — к шагу 8

2. Берем голову списка r из R (при этом список R уменьшается). Если r является макроправилом, то переходим к шагу 3, иначе — к шагу 4

3. $M = M \cup r$ (добавляем макроправило в M). Перейти к шагу 1

4. Обходим правую часть правила r и для каждого вызова некоторого макроправила $m \in M$ выполняем шаг 5. После замены всех вызовов макроправил добавляем обработанное правило r в R' (если в правой части правила r вызовов макроправил не было, то оно не изменится). Перейти к шагу 1

5. Если ранее встречался вызов макроправила m с теми же аргументами, то переходим к шагу 6, иначе — к шагу 7

6. Заменяем вызов макроправила m вызовом уже созданного правила. Возвращаемся к шагу 4

7. Создаем новое правило r' с уникальным именем. Правило r' получается из макроправила m путем замены формальных параметров фактическими. Для правила r' выполняем шаг 4. Добавляем r' в множество R'. Заменяем вызов макроправила m вызовом правила r'. Возвращаемся к шагу 4

8. Вернуть R' и закончить алгоритм.

Мы последовательно обрабатываем все правила нашей грамматики, добавляя макроправила в отдельное множество M (которое реализовано с помощью хэш-таблицы), а остальные — проверяя на наличие вызовов макроправил. Только обнаружив вызов макроправила $m \in M$, мы запускаем процесс его раскрытия. Сначала осуществляется проверка, не встречался ли ранее вызов макроправила m с теми же аргументами (для реализации такой проверки хранится специальная таблица ссылок). Если да, то текущий вызов параметризованного правила заменятся вызовом ранее сгенерированного правила, иначе вызовом нового параллельно создаваемого правила. При этом мы добавляем запись в таблицу ссылок еще до того, как сгенерировали новое правило. Это необходимо для обработки рекурсивного вызова макроправила с теми же аргументами.

Генерация нового правила происходит на основе параметризованного следующим образом: для него создается уникальное имя, затем мы обходим определение макроправила, заменяя все формальные параметры на фактические, тем самым получая искомое правило. Если на данном этапе мы встречаем вызов макроправила, то снова применяем к нему описанный выше алгоритм. В итоге, после работы алгоритма мы получаем множество R', не содержащее ни одного макроправила и состоящее из всех обычных правил множества R (у которых все обращения к макроправилам заменены) и правил, созданных в ходе выполнения алгоритма.

Докажем конечность описанного алгоритма:

1. общее число правил в грамматике конечно (а число необработанных постоянно уменьшается);

2. число обращений к макроправилам также конечно, то есть требуется сгенерировать конечное число правил для их раскрытия;

3. поскольку число аргументов макроправила конечно, при рекурсивном вызове (в правой части) того же макроправила, но с другими аргументами, мы за конечное число шагов придем к вызову макроправила, который уже был и сохранен в таблице ссылок. Уточним и обоснуем сказанное. Пусть в правой части

макроправила $m \in M$ происходит обращение к m, но с другими аргументами. Например, m имеет вид

$$m << p_1\, p_2 \ldots p_n >> : \alpha \,|\, m << p'_1\, p'_2 \ldots p'_n >>;$$

Аргументы m в правой части могут быть двух типов — те, которыми параметризовано макроправило m (назовем их макропараметрами — это $p_1\, p_2 \ldots p_n$) и обычные правила грамматики (назовем их статическими параметрами). Если p'_i (где $i \in [1..n]$) является статическим параметром, то на всех следующих шагах алгоритма раскрытия m он будет совпадать с макропараметром p_i, то есть p'_i совпадет с p_i для некоторого $i \in [1..n]$. Заметим, что когда для любого $i \in [1..n]$ p'_i совпадет с p_i, то мы получим вызов макроправила m с теми же аргументами, заменим его на вызов создаваемого правила и закончим алгоритм раскрытия макроправила m. Таким образом, если среди $p'_1\, p'_2 \ldots p'_n$ есть $k \in [1..n]$ статических параметров, то происходит фиксация этих k аргументов, и нам остается зафиксировать оставшиеся $n - k$ аргументов, то есть можно считать, что мы получаем макроправило с $n - k$ параметрами, причем все они являются макропараметрами. Поэтому для удобства будем считать, что $k = 0$ и для любого $i \in [1..n]$ p'_i — макропараметр. Тогда возможны два случая. Первый, $p'_1\, p'_2 \ldots p'_n$ является некоторой перестановкой $p_1\, p_2 \ldots p_n$. Поскольку число перестановок длины n конечно (это $n!$), то через конечное число шагов мы получим вызов m с перестановкой аргументов, которая уже встречалась (все вызовы у нас хранятся в таблице ссылок). Второй случай, p_i совпадает с несколькими p'_j, то есть p_i совпадает с $k \in [1..n]$ аргументами $p'_{j_1}, \ldots, p'_{j_k}$. При этом на каждом шаге k либо растет (и тогда на некотором шаге мы получим $k = n$), либо остается постоянным, тогда остальные $n - k$ аргументов образуют перестановку и доказательство сводится к первому случаю.

Из пунктов 1-3 следует, что предложенный алгоритм конечен.

Примеры работы алгоритма Рассмотрим результат работы алгоритма раскрытия макроправил на примере:

```
binExpr<<operand bOp>> : operand bOp operand ;

binOperator : PLUS | MINUS ;

simpleExpr : binExpr<<NUMBER binOperator>> ;
```

В этом случае будет сгенерирована грамматика (в нотации Elkhound) вида:

```
nonterm yard_binExpr_1 {
  -> NUMBER binOperator NUMBER ;
}

nonterm binOperator {
  -> PLUS ;
  -> MINUS ;
}
```

```
nonterm simpleExpr {
  -> yard_binExpr_1 ;
}
```

Повторный вызов того же макроправила и с теми же аргументами будет заменен вызовом ранее созданного правила. Допустим мы расширили грамматику из последнего примера правилом вида:

```
simpleAssignment : binExpr<<NUMBER binOperator>> ;
```

Тогда в Elkhound-грамматике будет дополнительно сгенерировано правило:

```
nonterm simpleAssignment {
  -> yard_binExpr_1 ;
}
```

Внутри параметризованных правил можно использовать конструкции расширенной формы Бэкуса-Наура и рекурсию. Например, правило `binExpr` можно переписать так (разрешив более сложные арифметические выражения):

```
binExpr<<operand bOp>> : operand ( bOp operand )* ;
```

Это правило можно переписать и с использованием рекурсии:

```
binExpr<<operand bOp>> : operand ( bOp binExpr<<operand bOp>> )? ;
```

При этом разрешается рекурсивно вызывать параметризованное правило не только с теми же аргументами (как в Menhir), но и с любыми другими. Например, следующее правило будет правильно преобразовано к обозначениям инструмента Elkhound:

```
binExpr<<operand bOp>> : operand ( bOp binExpr<<operand MULT>> )? ;
```

В качестве аргументов параметризованного правила могут выступать и другие макроправила (но без параметров), например:

```
binExpr<<unExpr operand bOp>>
: unExpr<<unaryOp operand>> ( bOp binExpr<<unExpr operand MULT>> )?
;
```

Таким образом, аргументами макроправила могут быть терминальные символы (лексемы), нетерминальные символы (названия правил) и другие параметризованные правила (но без аргументов).

3.3.3 Преобразование сгруппированных альтернатив

Elkhound не позволяет группировать альтернативы, то есть от следующего правила в обозначениях *YARD* нельзя напрямую перейти к нотации Elkhound:

```
someRule
: ( a | b ) c d ( e | f | g )
;
```

Поэтому на каждую группировку альтернатив (“`( a | b)`”, “`( e | f | g )`”) создается новое правило:

```
nonterm yard_alt_1 {
  -> a ;
  -> b ;
}

nonterm yard_alt_2 {
  -> e ;
  -> f ;
  -> g ;
}

nonterm someRule {
  -> yard_alt_1 c d yard_alt_2 ;
}
```

Каждая группировка альтернатив заменяется вызовом параллельно создаваемого правила. При этом даже на одинаковые группировки альтернатив, встречающиеся несколько раз, создаются различные правила. Это связано с тем, что трудно определить их равенство (поскольку также требуется проверять и семантические действия).

3.3.4 Преобразование атрибутов

Атрибуты делятся на синтезируемые (семантические действия) и наследуемые. Elkhound поддерживает только синтезируемые, другими словами спецификация грамматики разбираемого языка в Elkhound является S-атрибутной грамматикой [4].

При раскрытии конструкций расширенной формы Бэкуса-Наура, параметризованных правил и обработке сгруппированных альтернатив следует параллельно преобразовывать и атрибуты для того, чтобы грамматика сохраняла свою корректность и целостность.

Преобразование семантических действий При преобразованиях грамматики таких, как раскрытие конструкций расширенной формы Бэкуса-Наура, параметризованных правил и обработке сгруппированных альтернатив, сами семантические действия не меняются, а изменяется только их местоположение в грамматике. Уточним, как это происходит на примере сгруппированных альтернатив:

```
someRule : res=( a { action1 } | b { action2 } ) c d { myFunc res };
```

Здесь сгруппированная альтернатива (“`( a | b )`”) возвращает некоторое значение (`action1` или `action2`), которое сохраняется в переменной `res` (и далее передается в качестве аргумента в пользовательскую функцию `myFunc`). В обозначениях Elkhound правило `someRule` “превратится” в два (за счет выделения специального правила под сгруппированную альтернативу):

```
nonterm('yard_type_yard_alt_1) yard_alt_1 {
  -> a { action1 }
  -> b { action2 }
}

nonterm('yard_type_someRule) someRule {
```

```
  -> res:yard_alt_1 c d { myFunc res }
}
```

При этом семантические действия из сгруппированной альтернативы оказываются в новом правиле (`yard_alt_1`).

В Elkhound-грамматике, если правило возвращает некоторое семантическое значение, то надо явно указывать его тип. При генерации грамматики из нотации *YARD* тип семантического значения не известен, поэтому приходится использовать возможности языка OCaml — переменные типа (type variables). В последнем примере ими являются `'yard_type_yard_alt_1` и `'yard_type_someRule`. Они применяются для обозначения неизвестного типа, который будет вычислен компилятором OCaml.

Преобразование наследуемых атрибутов В Elkhound наследуемые атрибуты не поддерживаются, но за счет использования средств языка OCaml оказывается возможным осуществить такую поддержку. В OCaml функция может возвращать в качестве результата другую функцию, благодаря этому можно реализовать наследуемые атрибуты. В *GLR*-алгоритме (как впрочем и в *LR*-алгоритме) они не могут влиять на выполнение разбора, так как таблица состояний обычно является статической и создается еще до того, как будут проинициализированы наследуемые атрибуты. Таким образом, они влияют только на семантические действия, поэтому можно считать, что правило возвращает не значение, а функцию, которая зависит только от наследуемых атрибутов (и принимает их в качестве аргументов). Такой способ вычисления наследуемых атрибутов подробно рассмотрен в статье [17]. Рассмотрим реализацию наследуемых атрибутов на примере:

```
someRule[a b] : res1=c[a] res2=d[b res1] { (res1, res2) } ;
```

Правило `someRule` принимает два наследуемых атрибута — `a` и `b`. Первый из них передается в правило `c`, а второй и семантическое значение `res1` (которое вернуло после разбора правило `c`) передаются в качестве наследуемых атрибутов далее в правило `d`. Теперь посмотрим как будет выглядеть правило `someRule` в нотации Elkhound:

```
nonterm('yard_type_someRule) someRule {
  -> _elk_0:c _elk_1:d
     { fun a b ->
         let res1 = _elk_0 a in
         let res2 = _elk_1 b res1 in
       (res1, res2)
     }
}
```

Как видно из примера, чтобы реализовать наследуемые атрибуты пришлось несколько преобразовать, точнее дополнить семантические действия, и теперь правило `someRule` возвращает не пару из некоторых значений `res1` и `res2`, а функцию от атрибутов (функции возвращают и правила `c` и `d`). Но семантика при этом не изменилась, поскольку без значений атрибутов невозможно вычислить значения семантического действия, а при известных атрибутах мы, очевидно, получим одинаковые результаты.

В рассмотренном примере синтезируемый атрибут `res1` (правила `c`) выступает в качестве наследуемого атрибута правила `d`, такая "передача" атрибутов возможна только слева направо. Действительно, предположим, что синтезируемый атрибут `res2` (правила `d`) является наследуемым атрибутом для правила `c`:

```
someRule[a b] : res1=c[a res2] res2=d[b] { (res1, res2) } ;
```

В этом случае *YARD* преобразует исходное правило следующим образом:

```
nonterm('yard_type_someRule) someRule {
  -> _elk_0:c _elk_1:d
     { fun a b ->
         let res1 = _elk_0 a res2 in
         let res2 = _elk_1 b in
       (res1, res2)
     }
}
```

Теперь когда будет вызван компилятор языка OCaml , он выдаст сообщение об ошибке (символ `res2` не определен) при компиляции семантического действия. Тем самым описанная реализация гарантирует, что наследуемые атрибуты каждого символа X_i из правой части правила $A \to X_1 \dots X_n$ зависят только от атрибутов символов $X_1 \dots X_{i-1}$ и наследуемых атрибутов правила A. Следовательно (по определению), грамматика разбираемого языка в *YARD* задается с помощью L-атрибутной грамматики [4].

В статье [17] показано, что можно разрешить наследуемым атрибутам символа X_i из правой части правила $A \to X_1 \dots X_n$ зависеть от атрибутов символов $X_{i+1} \dots X_n$. Было решено не поддерживать такую возможность на уровне входного языка *YARD*, как ненадежную практику программирования. С другой стороны, используя технику, описанную в [17] в семантических действиях, можно фактически уйти от ограничений L-атрибутной грамматики. Покажем, как это сделать, на последнем примере. Переопределим правило `someRule` следующим образом:

```
c[a] : (* ... *) { fun x -> (* ... *) } ;

someRule[a b]
: res1=c[a] res2=d[b]
  { let res3 = res1 res2 in (res3, res2) }
;
```

Теперь в качестве наследуемого атрибута символа `c` выступает синтезируемый атрибут символа `d`, расположенного правее (в цепочке).

При раскрытии конструкций расширенной формы Бэкуса-Наура и обработке сгруппированных альтернатив следует параллельно преобразовывать и наследуемые атрибуты, чтобы грамматика сохраняла свою корректность.

Алгоритм преобразования наследуемых атрибутов одинаков в обоих случаях (и при раскрытии конструкций расширенной формы Бэкуса-Наура, и при обработке сгруппированных альтернатив). Он сводится к тому, что при выделении нового правила следует в него передавать (в качестве наследуемых атрибутов) не только наследуемые атрибуты, доступные в месте вызова создаваемого правила, но и все левые связывания (то есть все имена, участвующие в связываниях, расположенных левее вызова создаваемого правила). Пусть у нас есть правило вида

```
someRule[a b]
: res1=c[a] res2=d[b] res3=( e[a b res2] )? { (res1, res2, res3) }
;
```

В результате действия алгоритма (параллельно применяется алгоритм для раскрытия конструкций расширенной формы Бэкуса-Наура, описанный в части 3.3.1) правило `someRule` будет преобразовано следующим образом:

```
someRule[a b]
: res1=c[a] res2=d[b] res3=yard_option<<yard_item_1>>[a b res1 res2]
  { (res1, res2, res3) }
;

yard_item_1[a b res1 res2] : e[a b res2] ;
```

Как видно из примера, такое преобразование наследуемых атрибутов является промежуточным при переходе к нотации Elkhound .

3.3.5 Преобразование предикатов

Предикаты позволяют во время выполнения (*in run-time*) влиять на разбор входного потока. В *YARD* предикат — это логическое выражение, которое вычисляется во время разбора. Если выражение истинно, то разбор продолжается в обычном режиме, а если ложно, то разбор текущей альтернативы заканчивается, как если бы была обнаружена синтаксическая ошибка. Рассмотрим небольшой пример:

```
someRule
: a=INT "+" b=INT =>{ a + b < 239 }=> ( "*" INT )* { a + b }
| a=INT ( ( "+" | "*" ) INT )* { a }
;
```

Здесь с помощью предиката мы разрешаем неоднозначность в грамматике. Если сумма первых двух чисел меньше 239, то мы разбираем входной поток одним образом, в противном случае — другим.

В Elkhound-грамматике для каждого нетерминала можно определить функцию `keep`, которая должна возвращать `true`, если следует сохранить результат правила (его семантическое значение), в противном случае — `false` — тогда считается, что при разборе с использованием данного правила произошла ошибка. Благодаря этому оказывается возможным реализовать предикаты в Elkhound . Предварительно мы преобразуем *YARD*-грамматику, выделяя в отдельное правило последовательность терминалов и нетерминалов, расположенную слева от предиката (включая сам предикат и все связывания, которые используются внутри этой последовательности). При этом в качестве семантического значения новое правило возвращает кортеж из всех связываний. Для нашего примера описанное преобразование выглядит так

```
yard_predicate
: a=INT "+" b=INT =>{ a + b < 239 }=> { (a,b) }
;

someRule
: <(a,b)>=yard_predicate ( "*" INT )* { a + b }
| a=INT ( ( "+" | "*" ) INT )* { a }
;
```

Правило `yard_predicate` возвращает кортеж из всех связываний для того, чтобы они оставались видны правее предиката в исходном правиле. Тем самым обеспечивается корректность описанного преобразования грамматики. Теперь по правилу `yard_predicate` несложно сгенерировать соответствующее правило с функцией `keep` в нотации Elkhound:

```
nonterm('yard_type_predicate) yard_predicate {
  fun keep (v) {
    let (a,b) = v in ( a + b < 239 )
  }

  -> a:INT "+" b:INT { (a,b) }
}
```

Если предикат окажется истинным, то функция `keep` вернет `true`, и разбор входного потока продолжится, иначе анализатор будет считать, что произошла ошибка и попытается использовать другую альтернативу для разбора. Таким образом, семантика предиката при переходе к Elkhound-нотации сохранилась.

В данной реализации предикатов есть ограничение. Оно состоит в том, что в предикате нельзя использовать наследуемых атрибутов. Это связано с тем, что они становятся известны после операции свертки, а функция `keep` вычисляется до (поскольку от ее результата зависит, какое действие выполнит анализатор — сдвиг или свертку).

Если бы Elkhound позволял возвращать не единственный результат разбора, а список из всех результатов, тогда в предикатах можно было бы использовать наследуемые атрибуты, а вычисление предикатов откладывать до тех пор, пока не станут известны необходимые наследуемые атрибуты, после чего фильтровать список результатов. Мы "оставляем" только те результаты, которым соответствуют истинные резольверы. Если после такой фильтрации список результатов все равно содержит больше одного элемента, то в качестве результата разбора берется первый элемент списка. При этом важно, чтобы последовательность элементов в списке соответствовала порядку записи альтернатив в грамматике. Так для рассмотренного примера первая альтернатива (с предикатом) из правила `someRule` для нас "важнее" (поскольку записана раньше), чем вторая, поэтому в списке результат ее разбора всегда будет выступать в качестве первого элемента.

3.3.6 Генерация списка терминальных символов

Грамматика в нотации инструмента Elkhound содержит обязательную секцию `terminals`, в которой перечислены все терминальные символы (лексемы). В этой секции указаны и типы лексем — типы используются только при связывании значения лексемы с переменной (в Elkhound связывание выглядит так: `tokValue:MY_TOKEN`). Elkhound не разрешает осуществлять такое связывание для лексем, у которых тип не указан.

В грамматике для генератора *YARD* секции `terminals` (или эквивалентной ей) нет, поэтому при преобразовании *YARD*-грамматики к нотации Elkhound осуществляется генерация данной секции. При указании типов лексем используются переменные типа (см. часть 3.3.4).

Для того чтобы получить список терминальных символов, осуществляется обход дерева разбора грамматики, и все лексемы добавляются в список (при этом гарантируется, что никакая лексема не встречается в нем дважды). Например, если наша *YARD*-грамматика состоит из одного правила вида:

```
myExpr : NUMBER ( ( "+" | "-" ) NUMBER )* ;
```

то в Elkhound-грамматике будет сгенерирована следующая секция `terminals`:

```
terminals {
0 : YARD_EOF ;
1 : NUMBER ;
2 : LITERAL_2 "+" ;
3 : LITERAL_3 "-" ;

token('yard_token_1) NUMBER ;
token('yard_token_2) LITERAL_2 ;
token('yard_token_3) LITERAL_3 ;
}
```

Как видно из примера все лексемы пронумерованы начиная с нуля, причем нулю должен соответствовать терминальный символ для конца файла (таково требование инструмента Elkhound). Литералы в Elkhound являются "синонимами" для лексем, поэтому были сгенерированы терминальные символы `LITERAL_2` и `LITERAL_3`. После перечисления всех лексем в секции `terminals` определяются их типы.

4 Результаты

В ходе выполнения данной дипломной работы были получены следующие результаты:

- выполнен обзор современных инструментов автоматизации создания синтаксических анализаторов;
- сформулированы требования к генератору, которые появляются при решении задач автоматизированного реинжиниринга программ;
- предложен язык для задания грамматик анализируемых языков с учетом сформулированных требований;
- реализован прототип инструмента с предложенным входным языком и эффективным внутренним алгоритмом (*GLR*).

4.1 Пользовательский интерфейс инструмента *YARD*

Инструмент *YARD* предоставляет пользователю интерфейс командной строки. Пример использования:

```
yard [опции] [файл с грамматикой]
```

Опции инструмента *YARD* описаны в таблице 2.

Опция	*Описание*
-gen <генератор>	выбор одного из доступных генераторов (**glr, ostap, html** и другие)
-o <файл>	опция для задания выходного файла
-v	печать версии инструмента
-help	печать сообщения с кратким описанием опций

Таблица 2: Доступные опции инструмента *YARD*.

4.2 Дальнейшее развитие инструмента *YARD*

На уровне входного языка инструмента *YARD* (см. приложение А) предусмотрено наличие конструкций перестановки и расширенных регулярных выражений, необходимость которых обоснована в части 2.1, но в рамках данной работы эти конструкции не были реализованы. Другим развитием функциональности предложенного инструмента является поддержка модульной структуры грамматики. Возможность описания грамматики в нескольких файлах (модулях) является важной составляющей инструмента, предназначенного для промышленного использования (см. часть 2.1).

A Спецификация входного языка генератора *YARD*

```
grammar : [ action ] ( rule ";" )+ [ action ]

rule : ruleName [ paramList ] ":" expr ";"

ruleName : LIDENT ;

paramList : [ "<<" ruleName+ ">>" ] "[" PATT+ "]"

expr : alternative [ "|" alternative ]*

alternative : prefixed+ [ action ]

prefixed : [ "-" ] basic

basic : [ binding ] postfix [ predicate ]

postfix : primary [ "*" | "+" | "?" | continuation ] | permutation

continuation : "[" INTEGER [ "," INTEGER ] "]"

permutation : "[|" permItemList "|]"

permItemList : permItem [ "," permItem ]+

permItem : ruleName | UIDENT

primary : UIDENT | reference [ parameters ] | STRING | "(" expr ")"

reference : ruleName | "@" qualified

qualified : ruleName | UIDENT "." qualified

parameters : [ "<<" ruleName+ ">>" ] "[" EXPR "]"

binding : ( "<" PATT ">" | LIDENT ) "="

predicate : "=>{" EXPR "}=>"

action : "{" EXPR "}"
```

Здесь используются следующие обозначения:

- `UIDENT` и `LIDENT` — идентификаторы, начинающиеся с заглавной и прописной буквы соответственно;
- `INTEGER` — целое положительное число;
- `EXPR` — выражение в языке OCaml;

- `PATT` — образец (pattern) в языке OCaml ;
- `STRING` — строковый литерал.

B Грамматика входного языка генератора *YARD* в нотации *ocamlyacc*

```
%{
  open IL
  open IL.Production
  open IL
  open Misc
  let getList = function Some x -> x | None -> []
  let createSeqElem bnd omitted r check =
      { binding = bnd; omit = omitted; rule = r; checker = check }
%}

%token EOF
%token COLON
%token SEMICOLON
%token EQUAL
%token BAR
%token STAR
%token PLUS
%token QUESTION
%token LPAREN
%token RPAREN
%token DGREAT
%token DLESS
%token <IL.Source.t> STRING LIDENT UIDENT
%token <IL.Source.t> PREDICATE ACTION
%token <IL.Source.t> PATTERN PARAM

%start file
%type <(IL.Source.t, IL.Source.t) IL.Definition.t> file

%%

file
: actionOpt ruleList actionOpt EOF
  { { Definition.head = $1; Definition.grammar = $2; Definition.foot = $3 } }

actionOpt
: ACTION      { Some $1 }
| /* empty */ { None    }

ruleList
: rule SEMICOLON ruleList { $1::$3 }
| rule SEMICOLON          { [$1]   }

rule
: plusOpt LIDENT formalMacroParamOpt paramOpt COLON alts
```

```
  { { Rule.public = $1
    ; Rule.name = Source.toString $2
    ; Rule.macroArgs = getList $3
    ; Rule.body = $6
    ; Rule.args = o2l $4
    }
  }

plusOpt
: PLUS        { true  }
| /* empty */ { false }

formalMacroParamOpt
: DLESS formalMacroList DGREAT { Some $2 }
| /* empty */                  { None    }

formalMacroList
: LIDENT                 { [$1]   }
| LIDENT formalMacroList { $1::$2 }

paramOpt
: PARAM       { Some $1 }
| /* empty */ { None    }

alts
: seq            { $1            }
| seq barSeqList { PAlt ($1, $2) }

barSeqList
: BAR seq barSeqList { PAlt($2, $3) }
| BAR seq            { $2           }

seq
: seqElem seqElemList actionOpt { PSeq ($1::$2, $3) }
| ACTION                        { PSeq([], Some $1) }

seqElemList
: seqElem seqElemList { $1::$2 }
| /* empty */         { []     }

seqElem : bound predicateOpt { { $1 with checker = $2 } }

predicateOpt
: PREDICATE   { Some $1 }
| /* empty */ { None    }

bound
: patt EQUAL prim { createSeqElem (Some $1) false $3 None }
```

```
| prim            { createSeqElem None false $1 None      }

patt
: LIDENT  { $1 }
| PATTERN { $1 }

prim
: prim STAR           { PMany $1    }
| prim PLUS           { PSome $1    }
| prim QUESTION       { POpt $1     }
| LPAREN alts RPAREN  { $2          }
| call                { $1          }
| STRING              { PLiteral $1 }

macroParam
: LIDENT { $1 }
| UIDENT { $1 }
| STRING { $1 }

macroParams
: macroParam             { [$1]   }
| macroParam macroParams { $1::$2 }

macroParamOpt
: DLESS macroParams DGREAT { Some $2 }
| /* empty */              { None    }

call
: UIDENT { PToken $1 }
| LIDENT macroParamOpt paramOpt
  { match $2 with
    | None -> PRef  ($1, $3)
    | Some x -> PMacroRef ($1, $3, x)
  }
```

C Грамматика входного языка генератора *YARD* в нотации *YARD*

```
{ open IL
  open IL.Production
  open IL
  open Misc
  let getList = function Some x -> x | None -> []
  let createSeqElem bnd omitted r check =
      { binding = bnd; omit = omitted; rule = r; checker = check }
}

file
: h=ACTION? gr=( r=rule SEMI { r } )+ f=ACTION? EOF
  { { Definition.head = h; Definition.grammar = gr; Definition.foot = f } } ;

macroParams<<param>> : DLESS idlist=param+ DGREAT { idList } ;

rule
: p=PLUS? id=LIDENT margs=macroParams<<LIDENT>>? args=PARAM? COLON b=alts
  { { Rule.public = p <> None
    ; Rule.name = Source.toString id
    ; Rule.macroArgs = getList margs
    ; Rule.body = b
    ; Rule.args = o2l args
    }
  } ;

alts
: s=seq sl=( BAR se=seq { se } )*
  { List.fold_left (fun s e -> PAlt (s, e) ) s sl } ;

seq
: se=seqElem+ a=ACTION? { PSeq (se, a)      }
| a=ACTION              { PSeq ([], Some a) } ;

seqElem : b=bound pr=PREDICATE? { { b with checker = pr } } ;

bound
: pat=( p=patt EQUAL { p } )? pr=prim { createSeqElem pat false pr None } ;

patt
: id=LIDENT   { id  }
| pat=PATTERN { pat } ;

prim
: pr=prim
  r=( STAR { PMany pr } | PLUS { PSome pr } | QUESTION { POpt pr } ) { r }
```

```
| LPAREN b=alts RPAREN { b }
| c=call               { c }
| lit=STRING           { PLiteral lit } ;

macroParam
: id=LIDENT { id }
| id=UIDENT { id }
| id=STRING { id } ;

call
: id=UIDENT { PToken id }
| id=LIDENT mp=macroParams<<macroParam>>? p=PARAM?
  { match mp with
    | None   -> PRef (id, p)
    | Some x -> PMacroRef (id, p, x)
  } ;
```

Список литературы

[1] Автоматизированный реинжиниринг программ / Под ред. проф. А.Н. Терехова и А.А. Терехова. — СПб.: Издательство С.-Петербургского университета, 2000 — 332 с.

[2] Алгол 68. Методы реализации / Под ред. Г.С. Цейтина. — Л.: Изд-во Ленингр. ун-та, 1976. 224 с.

[3] *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. Т.1. Синтаксический анализ, 612 с.; Т.2. Компиляция, 487 с., М.: Мир, 1978.

[4] *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты. — М.: Издательский дом "Вильямс", 2003. — 768 с.

[5] *Макконнелл С.* Совершенный код. — СПб.: Питер, 2005. — 896 с.

[6] *Мартыненко Б.К.* Синтаксически управляемая обработка данных. — СПб.: Издательство С.-Петербургского университета, 1997. — 363 с.

[7] *Мартыненко Б.К.* Языки и трансляции. — СПб.: Издательство С.-Петербургского университета, 2002. — 229 с.

[8] *Чемоданов И.С., Дубчук Н.П.* Обзор современных средств автоматизации создания синтаксических анализаторов // Системное программирование. — СПб.: Изд-во С.-Петерб. ун-та, 2006. — с. 286–316.

[9] *Clark, Chris.* Conflicts // SIGPLAN Notices, Vol. 37, No. 8. — 2002. — p. 9–14.

[10] *Fokker, Jeroen.* Functional parsers // Lecture notes of the Baastad Spring school on functional programming. — 1995.

[11] *Hutton Graham, Meijer Erik.* Monadic parser combinators // Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham. — 1996.

[12] *Johnson S.C.* YACC — yet another compiler-compiler. // Bell Telephone Laboratories, Computing Science Technical Report No.32. — 1975.

[13] *Knuth, Donald.* Semantics of Context-free Languages // Mathematical Systems Theory, Vol. 2, No. 2. — 1968. — p. 127–145.

[14] *Mark G. J. van den Brand, A. Sellink, C. Verhoef* Current Parsing Techniques in Software Renovation Considered Harmful // IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension. — IEEE Computer Society, Washington, 1998.

[15] *McPeak, Scott.* Elkhound: A fast, practical GLR parser generator // University of California, Berkeley, Report No. UCB/CSD-2-1214. — 2002.

[16] *Robert D. Cameron.* Extending context-free grammars with permutation phrases // ACM Letters on Programming Languages and Systems, Vol.2, No. 1-4. — ACM Press, New York, 1993. — p. 85–94.

[17] *Thomas Johnsson.* Attribute grammars as a functional programming paradigm // Functional Programming Languages and Computer Architecture — 1987. — p. 154–173.

[18] *Tomita M.* An Efficient Augmented-Context-Free Parsing Algorithm // Computational Linguistics, 13(1-2). — 1987. — p. 31–46.

[19] *Wijngaarden A.* Revised Report on the Algorithmic Language ALGOL 68. // Acta Informatica, 5:1–236, 1974.

[20] Relativity Modernization Workbench User Manual. Available with the Modernization Workbench system ©Relativity Technologies (www.relativity.com).

[21] http://oops.tepkom.ru/projects/ostap (библиотека *Ostap*)

[22] http://caml.inria.fr (дистрибутивы и документация по языку OCaml)

[23] http://www.parsifalsoft.com (сайт разработчиков AnaGram)

[24] http://www.antlr.org (сайт разработчиков ANTLR)

[25] http://www.cwi.nl/projects/MetaEnv (сайт разработчиков ASF+SDF)

[26] http://www.gnu.org/software/bison (сайт разработчиков Bison)

[27] http://www.clearjump.com/products/clearparse/ (сайт разработчиков ClearParse)

[28] http://ssw.jku.at/Coco (сайт разработчиков Coco/R)

[29] http://tim.hec.ca/babin/CompTools/ (сайт разработчика CompTools)

[30] http://cppcc.sourceforge.net/ (сайт разработчика CppCC)

[31] http://www.cs.princeton.edu/~appel/modern/java/CUP/ (сайт разработчиков CUP)

[32] http://perso.ens-lyon.fr/emmanuel.onzon (сайт разработчика Dypgen)

[33] http://www.cs.berkeley.edu/~smcpeak/elkhound (сайт разработчиков Elkhound)

[34] http://grammatica.percederberg.net/index.html (сайт разработчиков Grammatica)

[35] https://javacc.dev.java.net (сайт разработчиков JavaCC)

[36] http://www.hwaci.com/sw/lemon/index.html (сайт разработчиков Lemon)

[37] http://www.cs.vu.nl/~ceriel/LLgen.html (сайт разработчиков LLGen)

[38] http://cristal.inria.fr/~fpottier/menhir (сайт разработчиков Menhir)

[39] http://vl.fmnet.info/precc/ (сайт разработчиков PRECC)

[40] http://www.cs.rhul.ac.uk/research/languages/projects/rdp.html (сайт разработчиков RDP)

[41] http://www.is.titech.ac.jp/~sassa/lab/rie-e.html (сайт разработчиков Rie)

[42] http://home.earthlink.net/~slkpg/index.html (сайт разработчиков SLK)

[43] http://www.speculate.de/styx/en/index.html (сайт разработчиков Styx)

[44] http://christophe.delord.free.fr/tpg/index.html (сайт разработчиков инструмента Toy Parser Generator)

[45] http://publib.boulder.ibm.com/infocenter/iadthelp/v6r0/topic/com.ibm.etools.iseries.langref.doc/c092539502.htm (онлайн-руководство по языку ILE Cobol)

[46] http://public.support.unisys.com/aseries/docs/ClearPath-MCP-11.0/PDF/86001047-507.pdf (справочное руководство по языку Unisys Work Flow Language)