Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Генератор синтаксических анализаторов для неоднозначных контекстно-свободных грамматик

Выпускная квалификационная работа бакалавра

студента 461 группы

Григорьева Семёна Вячеславовича

Научный руководитель	.... /подпись/	к.ф.-м.н. А.С. Лукичев
Рецензент	.... /подпись/	д.ф.-м.н., проф. Б.К. Мартыненко
"Допустить к защите" заведующий кафедрой	.... /подпись/	д.ф.-м.н., проф. А.Н. Терехов

Санкт-Петербург

2010

SAINT PETERSBURG STATE UNIVERSITY

Mathematics & Mechanics Faculty

Software Engineering Chair

Generator of parsers for arbitrary context free grammars with built-in support for EBNF

by

Grigoriev Semen Vjacheslavovich

Bachelor's graduation paper

Supervisor	.... /Signature/	PhD A.S. Lukichev
Reviewer	.... /Signature/	PhD, Prof. B.K. Martinenko
"Admitted to proof" Head of Chair	.... /Signature/	Professor A.N. Terekhov

Saint Petersburg
2010

Содержание

1 Введение **3**

2 Постановка задачи **6**

3 Основные определения **8**

4 Обзор **10**

4.1 Алгоритм анализа 10

4.2 Атрибутные грамматики. Подходы к вычислению атрибутов . 13

5 Реализация **15**

5.1 Алгоритм анализа. Основные функции 15

5.1.1 Поддержка расширенных контекстно-свободных грамматик 17

5.1.2 Построение леса вывода 18

5.2 Вычисление атрибутов 19

5.2.1 Задание семантических атрибутов в YARD 20

5.2.2 Построение LNFA по регулярному выражению 22

5.2.3 Построение LDFA по LNFA 25

5.2.4 Генерация кода для семантических действий пользователя 29

5.2.5 Интерпретация дерева вывода 30

5.3 Архитектура 32

6 Эксперименты **35**

6.1 Работа с однозначными грамматиками 35

6.2 Возможность работы с неоднозначными грамматиками . . . 37

6.3 Возможность работы с EBNF-грамматиками 40

6.4 Поддержка s-атрибутных грамматик 42

7 Заключение **46**

7.1 Дальнейшее развитие 46

1 Введение

Задачи автоматизированного реинжиниринга программ [1] выдвигают особые требования [12] к генераторам синтаксических анализаторов. Во многом это связано с тем, что теория синтаксически управляемой трансляции развивалась одновременно языками, сейчас называемых устаревшими (legacy languages). Тогда еще не были получены основные теоретические результаты, положенные в основу наиболее распространенных современных генераторов синтаксических анализаторов. Поэтому устаревшие языки имеют особенности, затрудняющие синтаксический анализ даже с использованием современных инструментов (например ASF+SDF [26], Elkhound [30]), которые значительно упрощают создание анализаторов.

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 после этого она серьёзно усложняется и на её сопровождение требуется больше ресурсов [12]. Поэтому устаревший язык обычно задается с помощью неоднозначной контекстно-свободной грамматики.

При решении задач реинжиниринга часто требуются преобразования уже существующей грамматики. С одной стороны, при разработке грамматики могут быть допущены неточности, с другой, документация устаревших языков и, в особенности, их диалектов может содержать ошибки, быть неполной или вообще отсутствовать. В результате, в целевом инструменте появляются ошибки, многие из которых возможно выявить только на этапе его тестирования. Для их исправления необходимо корректировать исходную грамматику. При этом, зачастую, изменение одного правила приводит к появлению десятков конфликтов в грамматике [12], которые необходимо разрешать „вручную“, что требует большого количества времени.

Кроме этого, на практике часто оказывается удобным иметь описание нескольким диалектов одного языка в одной грамматике. Это позволяет переиспользовать общие части грамматики, так как диалекты, как правило, имеют много общего. В то же время диалекты имеют характерные синтаксические конструкции, которые позволят автоматически определять

принадлежность входной строки тому или иному диалекту. Однако при таких описаниях часто возникают конфликты, которые так же приходится разрешать „вручную“.

Для решения этих задач предлагается использовать неоднозначные контекстно-свободные грамматики и соответствующие инструменты построения анализаторов [12]. Основная особенность этих инструментов – алгоритм, который способен работать с неоднозначными грамматиками (GLR-алгоритм). Анализатор, построенный по неоднозначной грамматике с помощью данного алгоритма, в общем случае, в результате разбора строит не единственное дерево, а несколько деревьев – лес, содержащий все возможные варианты вывода. Дальнейшая работа с полученным лесом организуется исходя из требований и особенностей решаемой задачи. Это избавляет он необходимости „ручного“ устранения конфликтов, что существенно сокращает время и упрощает разработку грамматики. Важным плюсом является ещё и то, что код становится более компактным и сопровождаемым.

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.

Удобным способом определения грамматики языка программирования является расширенная форма Бэкуса-Наура(EBNF) [9] – правила в такой грамматике в правых частях содержат регулярные выражения. На практике, использование EBNF позволяет упростить и сократить описание языка, сделать его более понятным. Кроме того, документация по языку, как правило, содержит конструкции EBNF. Однако многие современные инструменты не поддерживают EBNF-конструкции.

При работе с инструментом пользователь ожидает получить результат, описанный в терминах заданной им грамматики. Это выдвигает дополнительные требования к алгоритму. В случае, если входная грамматика была каким-либо образом преобразована, например с целью раскрыть конструкции EBNF, то появляется необходимость в построении „обратного“ преобразования. Это преобразование должно „перевести“ результат обратно в термины входной грамматики. Такие преобразования требуют дополни-

тельных ресурсов и усложняют инструмент. Поэтому наиболее предпочтительными является алгоритмы, работающие без дополнительных преобразований грамматики.

В рамках данной работы ставится цель разработки прототипа генератора анализаторов, позволяющего работать с неоднозначными расширенными контекстно-свободными грамматиками, предоставляющего, в то же время, ставшие привычными средства реализации трансляции, такие как атрибуты.

2 Постановка задачи

Цели данной работы:

- исследовать применимость рекурсивно-восходящего алгоритма для непосредственной поддержки EBNF-грамматик;
- исследовать особенности вычисления атрибутов при работе с расширенными произвольными контекстно-свободными грамматиками;
- реализовать прототип генератора синтаксических анализаторов, основанный на рекурсивно-восходящем алгоритме и показывающий возможность решения следующих задач:
  - работа с неоднозначными КС грамматиками
  - работа с EBNF-грамматиками без их преобразования
  - работа s-атрибутными грамматиками

Для их достижения необходимо решить следующие алгоритмические задачи:

- получить множество всех деревьев вывода для данной неоднозначной EBNF-грамматики для данной входной строки таким образом, чтобы это не требовало изменений входной грамматики;
- построить функции вычисления атрибутов, основанные на атрибутах входной грамматики, при условии, что входная грамматика является s-атрибутной;
- реализовать механизм вычисления функции вычисления атрибутов в конкретном узле дерева вывода, при условии, что оно построено для EBNF-грамматики;
- реализовать обход леса вывода для вычисления s-атрибутов;

Также необходимо показать следующее:

- предложенный алгоритм построения деревьев вывода имеет линейную временную сложность если на вход подана однозначная грамматика;

- предложенный алгоритм вычисления s-атрибутов корректно работает при наличии атрибутов с побочными эффектами;
- предложенный алгоритм вычисления s-атрибутов корректно работает с EBNF-грамматиками.

3 Основные определения

Определим ряд понятий и введём некоторые обозначения, необходимых для дальнейшего изложения.

Определение 1. *Конструкции регулярнх выражений*: будем говорить, что грамматика (правило) содержит конструкции регулярных выражений, если в записи правых частей правил используются элементы синтаксиса регулярных выражений (альтернатива, замыкание).

Определение 2. *Раскрытие конструкций регулярных выражений*: будем называть раскрытием конструкций регулярных выражений такое преобразование грамматики (правила), при котором исходная грамматика заменяется на эквивалентную, не содержащую конструкций регулярных выражений.

Определение 3. *Дерево вывода строки в EBNF-грамматике*: упорядоченное помеченное дерево D называется деревом вывода в EBNF-грамматике $G(S) = (N, T, P, S)$, если выполнены следующие условия:

1. корень дерева D помечен S;
2. каждый лист помечен либо $a \in T$, либо ε;
3. каждая внутренняя вершина помечена нетерминалом;
4. если N – нетерминал, которым помечена внутренняя вершина и $X_1, ..., X_n$ - метки ее прямых потомков в указанном порядке, то существует правило $N \rightarrow Y_1...Y_n \in P$ такое, что строка $X_1...X_n$ порождается регулярным выражением $Y_1...Y_n$.

Определение 4. *Непосредственная поддержка EBNF-грамматик*: будем говорить, что инструмент непосредственно поддерживает EBNF-грамматики, если он работает с ними без раскрытия конструкций регулярных выражений.

Определение 5. *Побочный эффект*: будем говорить, что функция или атрибут обладают побочным эффектом, если в процессе их вычислений возможно читать и модифицировать значения глобальных переменных, осу-

ществлять операции ввода/вывода, реагировать на исключительные ситуации, вызывать их обработчики.

Для примеров псевдокода, приводимых далее, будем использовать синтаксические соглашения, принятые в языке программирования F#.

4 Обзор

4.1 Алгоритм анализа

Одно из основных требований к инструментам автоматической генерации анализаторов, применяемым при решении задач реинжиниринга – возможность работать с неоднозначными грамматиками, так как с их помощью часто описываются устаревшие языки.

Существует несколько подходов к реализации алгоритма, позволяющего работать с грамматиками, содержащими неоднозначности:

- **алгоритм Томиты** (GLR-алгоритм) [8], основанный на организованном в виде графа стеке, что позволяет переиспользовать результаты вычислений и добиться хорошей производительности: $O(n)$ для однозначных грамматик и $O(n^3)$ в худшем случае;

- **алгоритм Эрли** (Early) [8], основа которого – итеративное повторение операций *prediction*, *scanning*, *completion* над специальным образом определённым состоянием. Пусть $S(k)$ множество состояний при позиции во входной строке k. Работа начинается с $S(0)$, состоящего только из начального правила. Операции *prediction*, *scanning*, *completion* можно определить следующим образом:

  – *prediction*: для каждого состояния из $S(k)$ вида $(X \rightarrow \alpha.Y\beta, j)$ добавить $(Y \rightarrow .y, k)$, для каждой продукции вида $(Y \rightarrow y)$

  – *scanning*: если a – следующий символ входной строки, то для каждого состояния из $S(k)$ вида $(X \rightarrow \alpha.a\beta, j)$ добавить $(X \rightarrow \alpha a.\beta, j)$ в $S(k+1)$

  – *completion*: для каждого состояния из $S(k)$ вида $(X \rightarrow y., j)$ найти $S(j)$ вида $(Y \rightarrow \alpha.X\beta, i)$ и добавить $(Y \rightarrow \alpha X.\beta, i)$ в $S(k)$.

  Имеет квадратичную сложность для однозначных грамматик и кубическую в общем случае.

- **рекурсивно-восходящий алгоритм** (recursive-ascent) [10] [11], основанный на наборе взаимно-рекурсивных функций, эмулирующих пе-

реходы между состояниями, и механизме запоминания результатов предыдущих вычислений;

- **CYK** [8], основанный на возможности использовать ранее построенные структуры и результаты.

- **Unger** [8], основную идею которого рассмотрим на простом примере. Пусть есть правило $S \rightarrow AB$ и строка abc. Надо решить задачу вывода этой строки из S. Для этого существует несколько возможностей:

  – из A выводится ab и из B выводится ;

  – из A выводится a и из B выводится b;

  Для каждой из этих возможностей надо аналогичным образом решить вопрос о выводимости.

Наиболее широко на практике применяется алгоритм Томиты. Алгоритм Эрли менее распространён. Алгоритмы CYK и Unger мало используются в основном из-за плохой с производительности [8].

В настоящее время наиболее популярным в практическом применении является алгоритм Томиты. Существует ряд инструментов, основанных на этом алгоритме:

- ASF+SDF [26] (Algebraic Specification Formalism + Syntax Definition Formalism) – генератор с широкими возможностями, но достаточно сложным входным языком. Является SGLR-инструментом (Scannerless, Generalized-LR).

- Bison [23] – 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).

- Elkhound [30] – позиционируется как быстрый и удобный GLR-инструмент, созданный в университете Беркли (США), однако обладает достаточно „бедным“ входным языком.

- DMS [20] – инструментарий „DMS Software Reengineering Toolkit“ включает в себя парсер генератор, основанный на GLR алгоритме.

- Happy [25] – парсер генератор с целевым языком Haskell [24]. Формат описания входной грамматики очень похож на формат классического YACC.

- Dypgen [17] – GLR-инструмент, обладающий такими особенностями как возможность удалять и добавлять правила во время синтаксического анализа, специфический способ задания приоритетов операций.

Все вышеперечисленные инструменты реализуют механизм анализа „сдвиг-свёртка“. Так же существует ряд других инструментов, основанных на том же алгоритме: APaGeD [15], DParser [16], eu.h8me.Parsing [18], GDK [21], SmaCC [19], Tom [22], UltraGram [31], Wormhole [28]. Все они реализуют GLR или SGLR алгоритм и ни один из них не реализует непосредственной поддержки EBNF-грамматик.

Интересующий нас рекурсивно-восходящий алгоритм реализуется на текущий момент только одним инструментом: Jade [14]. Jade – это генератор рекурсивно-восходящих LALR(1) парсеров с целевым языком C. При реализации данного инструмента возникла серьёзная проблема, связанная с большим объёмом кода целевого парсера. Так как при построении детерминированного парсера необходимо генерировать процедуры для каждого состояния, то объём кода быстро растёт с ростом количества правил в грамматике. Так, например, для языка Java объём кода составляет примерно 4 мегабайта [14]. В Jade эта проблема решается путём создания глобальной структуры(массива состояний), где хранится информация, позволяющая переиспользовать процедуры.

Однако существует подход к реализации рекурсивно-восходящего алгоритма, позволяющий решить проблему объёма кода [13]. С использованием этого подхода можно получить алгоритм для недетерминированного анализа, основанный всего на двух взаимно-рекурсивных функциях и позволяющий реализовать непосредственную поддержку EBNF-грамматик. Ещё одной особенностью является то, что проблемы определения левой границы отрезка в стеке, соответствующего текущему правилу, в данном подходе не

существует, так как стек вызовов рекурсивных функций хранит информацию о начале анализа по правилу [8].

4.2 Атрибутные грамматики. Подходы к вычислению атрибутов

При работе с неоднозначными грамматиками выдвигаются особые требования к алгоритму вычисления атрибутов. Это связано с тем, что в качестве атрибута пользователь может указать действие, обладающее побочным эффектом (например, печать на экран). При наличии таких атрибутов нельзя проводить вычисления непосредственно в процессе анализа, так как в момент разбора не возможно определить, завершиться ли текущая ветвь удачно. В ситуациях, когда при непосредственном вычислении ветвь завершилась неудачно, могут быть совершены лишние действия (например, лишняя печать на экран).

Были рассмотрены два подхода к решению этой проблемы:

- **Отложенные вычисления** (continuation passing style, CPS [7]). Непосредственно во время разбора атрибуты не вычисляются. Вычисления откладываются. Строится функция, которая вычисляется только один раз, после удачного завершения разбора.

- **Интерпретация леса вывода** - построение леса вывода и последующее вычисление атрибутов над ним. Первым шагом строится лес вывода, который содержит только деревья, соответствующие успешным вариантам разбора. Следующим шагом над полученным лесом производятся вычисления, соответствующие заданным атрибутам. При корректном задании пользовательских атрибутов (это должны быть s-атрибуты) деревья обходятся снизу-вверх и этот обход конечен. Данный подход является классическим, однако на практике во многих инструментах (например Bison) дерево вывода непосредственно не строится, а пользовательские атрибуты связываются с действиями LR-автомата.

Оба этих подхода гарантируют, что будут выполнены действия, соответствующие только успешным вариантам разбора. Однако второй подход

является более удобным для конечного пользователя, так как позволяет явно получить дерево вывода, что упрощает отладку. Именно он и был выбран для реализации.

5 Реализация

Для реализации выбран рекурсивно-восходящий алгоритм, модернизированный для работы с расширенными контекстно-свободными грамматиками без их преобразования.

В качестве фронтенда, обладающего мощным и удобным языком спецификации трансляции, был выбран инструмент YARD [5]. Он позволяет получить дерево разбора грамматики, заданной пользователем, которое используется в дальнейшем.

Выбранный подход к вычислению атрибутов – интерпретация дерева вывода. Основная идея заключается в том, что генератор строит набор функций, каждая функция соответствует одному правилу грамматики. После построения дерева вывода, оно обходится снизу вверх и в каждом узле вычисляется соответствующая ему функция. Функции вызываются по рефлексии, что показывает возможность в будущем сделать инструмент более гибким благодаря возможности динамически (в процессе работы анализатора) изменять функции вычисления атрибутов (это возможно благодаря возможностям динамической перекомпиляции, предоставляемыми языком реализации).

Инструмент реализован на платформе .NET [27], на функциональном языке F# [29].

Более подробно детали реализации описаны ниже.

5.1 Алгоритм анализа. Основные функции

Рекурсивно-восходящий алгоритм позволяет с помощью набора взаимно-рекурсивных функций эмулировать поведение LR-автомата. Его можно рассматривать, как аналог рекурсивного спуска, но для LR грамматики. Действительно, стек автомата естественным образом заменяется на стек вызова функций, а вызовы функций заменяют переходы автомата.

При таком подходе возникает проблема с объёмом кода целевого инструмента. Связана она с тем, что количество функций, которые необходимо построить, сравнимо с размером таблицы переходов LR-автомата, которые на практике бывают очень большими.

Для решения этой проблемы можно воспользоваться подходом, предложенным в работе [11]. Основная идея предложенного подхода – реализовать всего две взаимно-рекурсивные функции, но оперирующие при этом уже не одним состоянием, а множеством состояний. Важно, что при такой реализации можно получить недетерминированный анализ. При этом ветвление реализуется, как ветвление в одной из функций, а механизм для переиспользования результатов вычисления, слияние, можно реализовать, как запоминание результатов вычисления функций. Для этого можно воспользоваться такой конструкции функционального программирования, как замыкание.

В основе этого алгоритма лежат две взаимно-рекурсивные функции *parse* и *climb*, которые можно определить следующим образом:

- parse q i $=\{(A \to a., i) | A \to a. \in q\} \bigcup$

  $\{(A \to a.b, k) | i = xj, (A \to a.b, k) \in climb$ q x j$\} \bigcup$

  $\{(A \to a.b, k) | B \to e, (A \to a.b, k) \in climb$ q B j $\}$

- climb q X i $= \{(A \to a.Xb, k) | (A \to aX.b, k) \in parse($goto q X$)i, a \neq e, A \to a.Xb \in q\} \bigcup$

  $\{(A \to a.b, l) | (C \to X.c, j) \in parse($goto q X$)i, (A \to a.b, l) \in climb$ q C j$\}$

Где:

- $G = (V_T, V_N, P, S)$: контекстно-свободная грамматика;
- $V = V_T \cup V_N$;
- $x, y \in V_T$
- $i, j \in V_T^*$
- $X, Y \in V$
- $A, B, C \in V_N$
- $a, b, c \in V^*$
- q: LR-состояние (core)

- *goto q X* $= \{A \rightarrow aX.b | A \rightarrow a.Xb \in$q*},

- $q*$: замыкание. q* $= q \bigcup \{B \rightarrow .c | A \rightarrow a.Bb \in$q*$\} \bigcup \{x \rightarrow .x | A \rightarrow a.xb \in$q*} [2]

Далее, на основе этого алгоритма надо получить алгоритм, реализующий:

1. непосредственную поддержку EBNF-грамматик;

2. построение леса вывода.

Для того в него необходимо внести некоторые изменения, описанные далее.

5.1.1 Поддержка расширенных контекстно-свободных грамматик

Поддержка регулярных выражений в правых частях правил (EBNF-грамматики) получается естественным образом. Для этого правая часть правила представляется как детерминированный конечный автомат. LR-ситуация в таком случае может быть представлена парой: правило (нетерминал+КА) и номер состояния (соответствует позиции маркера в классическом определении).

Действительно, в правой части правила всегда находится регулярное выражение. В простом случае, когда это последовательность терминалов и нетерминалов, позиция маркера из классического определения LR-ситуации тривиальным образом соответствует состоянию конечного автомата, построенного по этой последовательности, а перемещение маркера – переход КА из одного состояния в другое. В общем случае по регулярному выражению из правой части правила по алгоритму Томпсона [2] строится недетерминированный конечный автомат (НКА). Заметим, что в полученном НКА много ε-переходов. Например, каждая альтернатива вносит два дополнительных состояния и 4 ε-перехода. Чтобы уменьшить количество переходов в результирующем LR-автомате можно преобразовать НКА в детерминированный конечный автомат(ДКА). Для этого применим стандартный алгоритм преобразования НКА в ДКА [2]. После этого заменим позицию маркера номером состояния полученного ДКА.

Таким образом, мы можем строить LR-ситуации для EBNF-грамматик. Для каждой, построенной LR-ситуации, для дальнейшей работы необходимо хранить следующую информацию:

- номер правила;
- левую часть правила(нетерминал);
- номер текущего состояния ДКА;
- символ принимаемый ДКА в данном состоянии;
- номер состояния ДКА, в которое он перейдёт приняв данный символ;
- номер начального состояния ДКА;
- номера конечных состояний ДКА;

Функция *goto* для работы с новыми LR-ситуациями основана на стандартном алгоритме вычисления GOTO при LR анализе [2]. Его необходимо лишь изменить таким образом, чтобы он мог работать с LR-ситуациями, определённого нами вида.

5.1.2 Построение леса вывода

Для того, чтобы построить лес вывода, необходимо добавить в функции механизм конструирования очередного узла дерева и предусмотреть сохранение леса.

В функции *parse* необходимо производить конструирование нового листа, в случае, когда происходит чтение очередного символа из входной цепочки.

В функции *climb* необходимо конструировать новый внутренний узел из поддеревьев, в случае, когда нужно произвести свёртку по текущему правилу, и объединять множества поддеревьев, в случае, если работа с текущем правилом ещё не закончена (сместился маркер в правой части).

Введём следующие обозначения:

- $A \to R$ – правило грамматики, где A – нетерминал, R – ДКА, построенный по регулярному выражению;

- $(A \to R, i)$ – LR-ситуация, где i – состояние ДКА;
- $is - final\ R\ i$ – функция , осуществляющая проверку, что i – конечное состояние R;
- $exists\ elem\ set$ – функция, проверяющая наличие элемента $elem$ в множестве set;
- $(leaf : a)$, $(A- > ...)$ – конструкции дерева разбора;

Тогда сигнатуры функции будут иметь следующий вид:

$parse\ q\ \{u|A \to a.b, u = vx, b \to v\} \to (A \to a.b)x\{b\})$

$climb\ q\ X\ \{u|A \to aX.b, u = vx, b \to v\}(tree$: синтаксическое дерево для X) $\to (A \to a.Xb)x\{Xb\}$

А сами функции будут выглядеть так:

```
parse q u =
    if exists (A → R,i) q & is-final(R,i)
    then (A → R,i;u;[])
    else
      if u=av & exists ( A →  R,i) q* & R(i,a)=j
      then climb q a v (leaf:a)
    else
    if exists (A → R,0) q* & is-final(R,0)
    then climb q A v (A → [])
climb q X u h =
     let (A → R,j;w;s) = parse (goto q* X) u in
     if R(i,X)=j & exists (A→  R,i) q
     then (A →  R,i;w;h::s)
     else climb q A w (A →  h::s)
```

5.2 Вычисление атрибутов

На практике для задания семантических действий применяются наследуемые (l-атрибутные грамматики) и вычислимые (s-атрибутные грамматики) атрибуты. В рамках данной работы была поставлена задача поддержать работу с s-атрибутными грамматиками.

Далее будут подробнее описаны особенности вычисления атрибутов при непосредственной поддержке EBNF-грамматик и алгоритм вычисления атрибутов.

5.2.1 Задание семантических атрибутов в YARD

Грамматика YARD-а [5] позволяет определять атрибуты для любой части продукции, которая является последовательностью. На практике это означает, что атрибут может быть ассоциирован не только с правилом целиком, а с любой его частью, которая является последовательностью. Например:

```
someRule : val1 = (a {action1} | b {action2})
           val2 = c  {someFunc val1 val2};
```

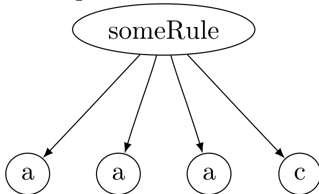
Здесь альтернатива `( a b )` возвращает некоторое значение (`action1` или `action2`), которое сохраняется в переменной `val1`, значение нетерминала сохраняется в переменной `val2`, и далее обе эти переменные передаются в качестве аргументов в пользовательскую функцию `someFunc`.

Возможность таким способом задавать атрибуты вызывает сложности при интерпретации дерева вывода. Связаны они с тем, что для вычисления атрибутов становится недостаточно информации только о дереве вывода входного выражения.

Рассмотрим эту проблему более подробно. Допустим, в грамматике есть правило:

```
someRule : val1 = (a {action1})* val2 = c  {someFunc val1 val2};
```

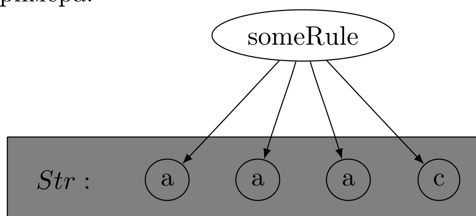
Узел дерева вывода, соответствующий правилу, приведённому выше, может выглядеть следующим образом:



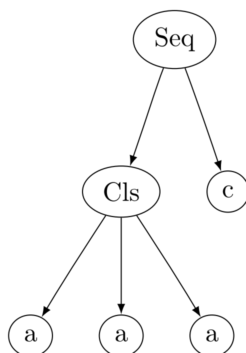
Для вычисления атрибутов в этом узле необходимо знать, что первые три сына были порождены из замыкания, их необходимо объединить в список и уже его передать в функцию `someFun` в качестве первого параметра.

В общем случае можно рассматривать непосредственных сыновей узла как строку, принадлежащую языку, задаваемому регулярным выражением в правой части правила. Тогда можно сказать, что для вычисления атрибутов нам необходимо дерево разбора этой строки.

Для нашего примера:



Где Str – строка, принадлежащая языку, задаваемому регулярным выражением в правой части правила. Дерево разбора этой строки будет выглядеть так:



Таким образом, для того, чтобы вычислять атрибуты, нам необходимо во время интерпретации дерева вывода входного выражения построить дерево разбора строки из сыновей узла, для которого непосредственно производятся вычисления. Для этого необходимо во время анализа поучить и сохранить информацию о выводе этой строки.

Удобным механизмом для решения этой проблемы оказался конечный автомат с помеченными переходами (далее будем для простоты будем называть их LFA – Labelled Finite Automaton). Общая схема решения такова:

1. строится недетерминированный конечный автомат с помеченными переходами (LNFA), в котором в качестве меток сохраняется информация о начале и завершении конструкций регулярного выражения, по которому строится этот LNFA;

2. по LNFA строится детерминированный конечный автомат с помечен-

ными переходами (LDFA);

3. в процессе анализа собираются и сохраняются метки с совершённых переходов и на основе этой информации строится дерево разбора.

Подробнее все эти шаги будут описаны далее.

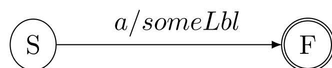
5.2.2 Построение LNFA по регулярному выражению

Определим LNFA как шестёрку $(Q, \Sigma, L, T, q_0, F)$, состоящая из:

- конечного множества состояний Q
- конечного множества входных символов Σ
- конечного множества меток L
- функции перехода $T: Q \times (\Sigma \cup \varepsilon) \to 2^{Q \times L}$
- начального состояния $q_0 \in Q$
- конечного множества финальных состояний $F \subseteq Q$

Таким образом переходы автомата снабжаются метками. При изображении автомата в виде графа переходам соответствуют рёбра. Будем записывать метки через знак "/" после символа, принимаемого автоматом при данном переходе.

Пример LNFA:



Где a – принимаемый символ, $someLbl$ – метка.

Чтобы решить задачу вычисления атрибутов необходимо знать, когда началось и когда закончилось распознавание той или иной конструкции регулярного выражения. Для этого определим метки специального типа:

- для обозначения начала и конца конструкции
  - лист: $LeafS$, $LeafE$;
  - последовательность: $SeqS$, $SeqE$;
  - замыкание: $ClsS$, $ClsE$;

– альтернатива: $Alt1S$, $Alt1E$, $Alt2S$, $Alt2E$ - пара меток для каждой ветви;

- ω – „пустая“ метка;

Метка для конкретного ребра будет состоять из типа метки и уникального идентификатора, который совпадает у меток начала и конца одной и той же конструкции. Таким образом, множество меток L можно определить так:

$$L = \{ \ t * k \mid t \in \{ \ LeafS, \ LeafE, \ SeqS, \ SeqE, \ ClsS, \ ClsE, \\ Alt1S, \ Alt1E, \ Alt2S, \ Alt2E, \ \omega \ \}, \ k \in N \ \} \qquad (1)$$

Для построения LNFA по регулярному выражению необходимо модернизировать алгоритм Томпсона. Его необходимо дополнить механизмом расстановки меток.

Будем рассматривать алгоритм, который работает с деревом вывода регулярного выражения, которое мы можем получить из YARD-а. Это дерево может содержать следующие конструкции:

- Leaf(a) – лист дерева. Соответствует символу в регулярном выражении.
- Seq(lst) – последовательность. lst – список элементов последовательности.
- Alt(L,R) – альтернатива.
- Cls(T) – замыкание.

Определим ряд функций:

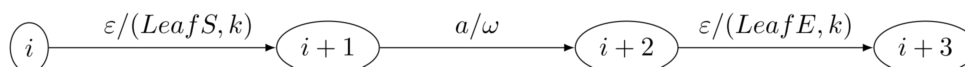
- $buildLNFA : \ 'Tree \rightarrow \ 'LNFA$ – строит LNFA по дереву вывода регулярного выражения.
- $map : \ ('T \rightarrow \ 'U) \rightarrow \ 'T \ list \rightarrow \ 'U \ list$ – применяет функцию, переданную в качестве первого аргумента, к каждому элементу списка, перданного вторым аргументом.

- *concat* : $'LNFA\ list \to 'LNFA$ – конкатенирует автоматы из списка, добавляя ε/ω-переходы.

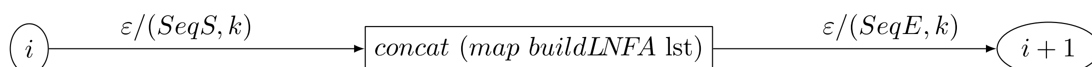
Так же предположим, что у нас есть функция для генерации уникального индекса k для нумерации меток.

Модернизированный алгоритм будет выглядеть следующим образом:

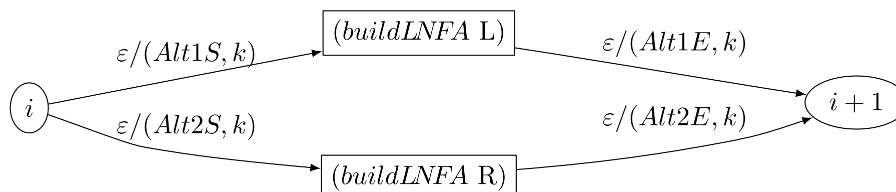
- Лист : Leaf(a)



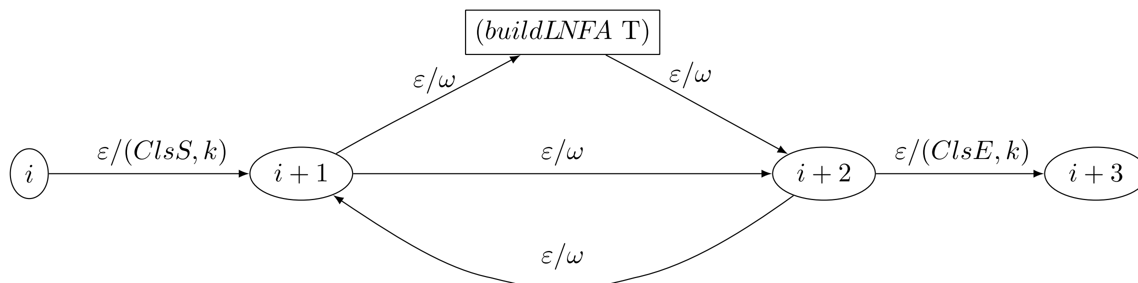
- Последовательность : Seq(lst)



- Альтернатива : Alt(L,R)



- Замыкание : Cls(T)



Таким образом, теперь мы можем по регулярному выражению построить LNFA с метками, соответствующими началу и концу каждой конструкции регулярного выражения.

5.2.3 Построение LDFA по LNFA

Для дальнейшего использования необходимо преобразовать недетерминированный автомат в детерминированный автомат. Для этого можно использовать стандартный алгоритм построения DFA по NFA [2], расширенный для работы с метками.

Определим детерминированный конечный автомат с метками (LDFA), как как шестёрку (Q, Σ, L', T, q_0, F), состоящая из:

- конечного множества состояний Q
- конечного множества входных символов Σ
- конечного множества меток L
- функции перехода $T: \ Q \times \Sigma \to Q \times L'$
- начального состояния $q_0 \in Q$
- конечного множества финальных состояний $F \subseteq Q$

В нашем случае $L' = 2^{(2^L)}$, где L – множество меток LNFA (1) и подмножества L являются упорядоченными.

Процесс построения детерминированного автомата с помеченными переходами можно разбить на 2 этапа:

1. построение детерминированного автомата с помощью стандартного алгоритма;
2. вычисление и расстановка новых меток.

Рассмотрим второй этап более подробно. Сперва необходимо разбить состояния DFA, построенного на первом шаге, на два множества: F – множество конечных состояний и $I = Q/F$ – остальные (не конечные) состояния.

Для дальнейшей работы нам понадобится функция $calculateNewLabel$: $'state \to' l$, где $'l \in L'$. Она будет по состоянию автомата вычислять метку

перехода из этого состояния. Определим эту функцию следующим образом:

$calculateNewLabel\ state\ =$

  $let\ e =$ множество всех ε-цепочек, заканчивающихся в state

  $let\ l = map\ (fun\ eLine \to$ пройти по $eLine$ из начала в конец

    и собрать по порядку все метки) e

  l

Основная функция для вычисления и расстановки новых меток:

$setNewLabel =$

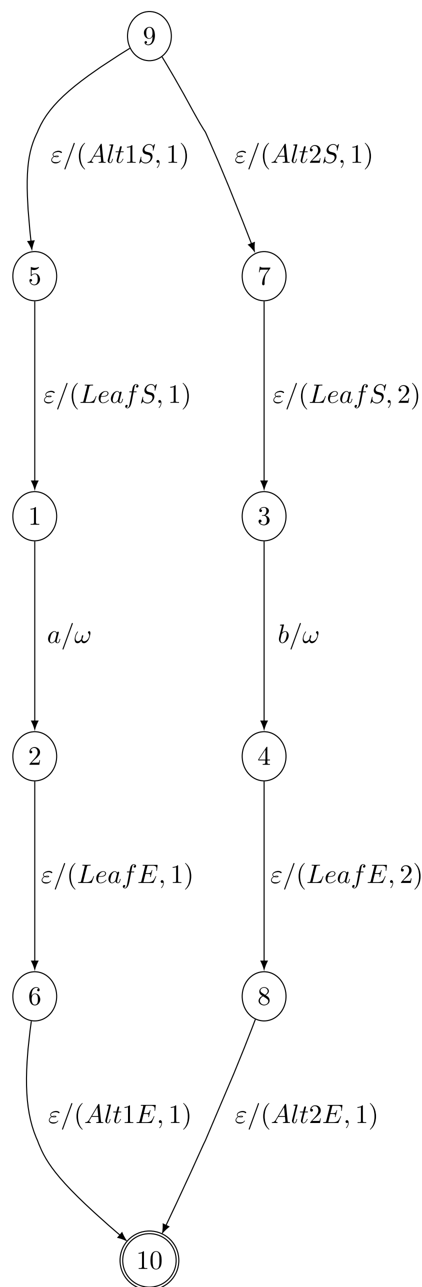
  Для каждого состояния $f \in F$ добавить новое „висячее“ ребро с началом в f

  и установить ему метку равную $(calculateNewLabel\ f)$

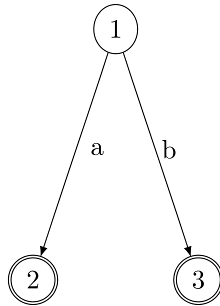
  Для каждого состояния $i \in I$, для каждого ребра, выходящего из i

  установить метку равную $(calculateNewLabel\ i)$

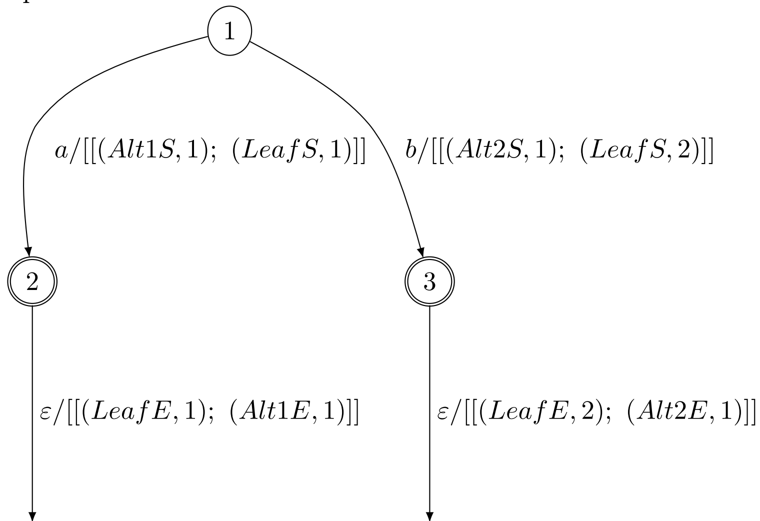
Рассмотрим работу данного алгоритма на примере. Пусть дано правило грамматики: $S \to a|b$. Необходимо построить ДКА с помеченными переходами для этого правила. По регулярному выражению в правой части ,применив алгоритм, описанный выше, построим НКА. В результате мы получим следующий автомат:



Далее, по этому автомату строится ДКА. Для построения применяется стандартный алгоритм Томпсона. Метки на рёбрах можно опустить, они будут вычислены на следующем шаге. Результирующий автомат будет выглядеть следующим образом:



После этого можно вычислить новые метки, способом описанным выше. Автомат изменится – в нём появятся две „висячие“ дуги – для каждого из финальных состояний. Выглядеть результирующий автомат будет следующим образом:



Видно, что метками в полученном детерминированном автомате являются множества списков меток недетерминированного автомата.

Теперь рассмотрим алгоритм, с помощью которого по построенному LDFA можно получить нужные нам данные о выводе входной строки.

1. *let LDFA* = LDFA, построенный по регулярному выражению, как описано выше.

2. *let* $labelsList = []$ (*список для хранения меток, полученных в процессе работы автомата*)

3. *let* $trace = []$ (*трасса работы автомата*)

4. Запускаем *LDFA* над входной строкой. При каждом переходе добавляем метку l_i в начало $labelsList$.

5. (*После завершения работы *LDFA* в $labelsList$ хранится инвертированная последовательность меток.*)

6. if строка принята

7. $then$

8. Просматриваем $labelsList$ из начала в конец.

9. $let\ l_i =$ текущий элемент $labelsList$

10. $let\ l_{i+1} =$ следующий элемент $labelsList$

11. Добавляем в $trace$ ($k \mid k \in l_i : \exists p \in l_{i+1}$: первый элемент из p является закрывающим для последнего элемента из k)

12. $else$

13. $Error$

14. $reverse\ trace$

Таким образом, мы построили детерминированный конечный автомат с помеченными переходами, с помощью которого можно получить необходимую для дальнейшей работы информацию о выводе входной строки.

5.2.4 Генерация кода для семантических действий пользователя

При задании грамматики пользователь может, пользуясь атрибутами, указывать семантические действия. Необходимо на основе этих атрибутов построить код, который будет производить необходимые вычисления.

Для вычисления атрибутов был выбран подход интерпретации дерева (леса) вывода. Основная идея этого алгоритма – обход заранее построенного дерева вывода и вычисление необходимых функций в его узлах.

В нашем случае дерево будет обходиться снизу вверх, так на данном этапе было решено поддержать только s-атрибутные грамматики.

При интерпретации дерева вывода оно обходится снизу-вверх и в каждом узле вычисляется некоторая функция, основанная на атрибутах из

пользовательской грамматики. Каждый внутренний узел в дереве вывода выражения в данной грамматике соответствует правилу из этой грамматики. Поэтому для интерпретации дерева достаточно построить функцию, соответствующую правилу грамматики и основанную на атрибутах, заданных в этом правиле. При обходе дерева в каждом его узле будет вычисляться соответствующая ему функция.

Функция всегда принимает один аргумент – дерево разбора строки из сыновей узла, в котором она вычисляется, и является интерпретатором этого дерева. Результат вычислений помещается в узел, для которого функция вычислялась.

Параллельно с генерацией кода строится таблица соответствий между функцией и правилом, для которого она построена. Эта таблица будет применяться при интерпретации дерева вывода для поиска функции, соответствующей текущему узлу дерева.

5.2.5 Интерпретация дерева вывода

Чтобы вычислить семантические действия, заданные пользователем, необходимо выполнить интерпретацию дерева вывода. Для этого нужно обойти дерево снизу вверх и в каждом внутреннем узле вычислить соответствующую функцию, которая ищется с помощью таблицы соответствий между правилом и функцией.

Общая схема интерпретации дерева выглядит следующим образом:

- построенное дерево разбора обходится снизу вверх;
- для очередного внутреннего узла, на основе трассы, хранимой в нём, строится дерево разбора строки из сыновей;
- с помощью таблицы, построенной на этапе генерации, ищется функция, соответствующая данному узлу;
- найденная функция применяется к построенному дереву разбора;
- результат сохраняется в текущем узле;

В процессе анализа в узлах дерева вывода накапливается информация, необходимая для построения дерева разбора строки из его сыновей в виде

трасс вычислений соответствующих автоматов.

Полученная трасса является, по сути своей, правильной скобочной структурой, которую надо наложить на строку из сыновей. Сделать это не сложно, так как каждый символ строки окружён скобками. После этого становится возможным построить дерево разбора строки сыновей. Для этого скобочная пара сворачивается в узел дерева, а все элементы, лежащие внутри скобок, превращаются в сыновей этого узла.

Дерево разбора будет содержать четыре типа узлов:

$type\ aTree <\ 'value > =$

$\quad | \ ASeq\ of\ List <aTree>$

$\quad | \ AAlt\ of\ Option <aTree> *Option <aTree>$

$\quad | \ ACls\ of\ List <aTree>$

$\quad | \ ALeaf\ of\ 'value$

Алгоритм построения дерева выглядит следующим образом:

$buildTree\ line =$

$\quad let\ group =$ первая скобочная пара из line

$\quad let\ end =$ line без group

$\quad let\ tree = match$ тип внешних скобок group $with$

$\qquad | \ Seq \rightarrow ASeq(buildTree$ значение внутри скобок group)

$\qquad | \ Alt1 \rightarrow AAlt((buildTree$ значение внутри скобок group), $None)$

$\qquad | \ Alt2 \rightarrow AAlt(None, (buildrTree$ значение внутри скобок group))

$\qquad | \ Cls \rightarrow ACls(buildTree$ значение внутри скобок group)

$\qquad | \ Leaf \rightarrow ALeaf($ значение внутри скобок group)

$\quad if\ end$ пусто

$\quad then\ [tree]$

$\quad else\ tree :: (buildTree\ end)$

Предложенный выше алгоритм, позволяет вычислять пользовательские атрибуты, обеспечивая при этом непосредственную поддержку EBNF-грамматик.

5.3 Архитектура

В ходе работы был реализован прототип генератора синтаксических анализаторов, основанный на описанном выше алгоритме. В процессе изучения алгоритма было выяснено, что его удобно реализовывать на функциональном языке программирования, поэтому прототип реализован на функциональном языке программирования для платформы .NET – F#. Общая схема реализованного инструмента приведена ниже.

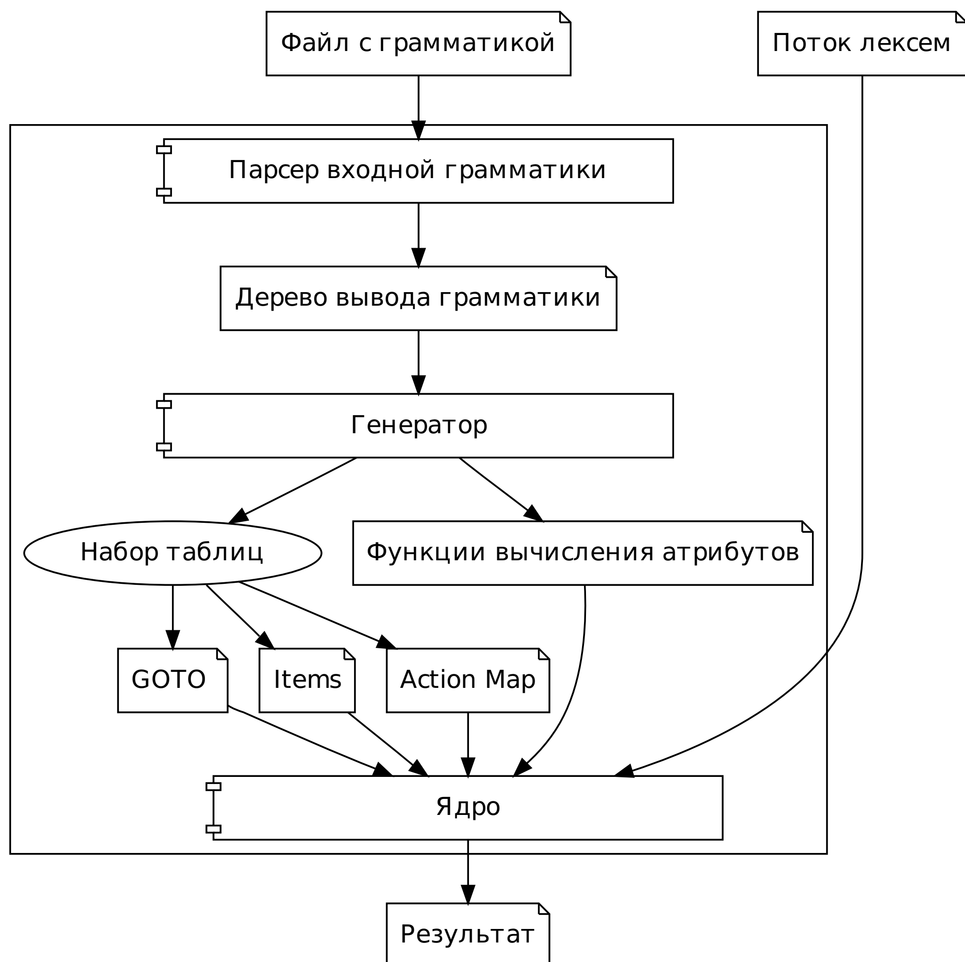


Рис. 1: Общая структура инструмента

Реализованный инструмент состоит из трёх основных модулей:

- **Парсер входной грамматики** (фронтенд), который основан на инструменте YARD, из которого используется входной язык (соответственно лексический и синтаксический анализаторы этого языка), внутреннее представление языка описания грамматики и набор преобразований (используется в генераторе).
- **Генератор**, который по дереву грамматики строит набор таблиц и генерирует функции для вычисления пользовательских атрибутов. Основные этапы:
  - **Преобразования грамматики**, которые необходимы, так как на уровне входного языка YARD поддерживает конструкции, которые на текущий момент наш инструмент не поддерживает(например макроправила).
  - **Генерация таблиц и кода**

  Результатами работы генератора являются:
  - **Набор таблиц**, который содержит данные, необходимые для синтаксического анализа и вычисления атрибутов и состоит из:
    - **GOTO** – таблица переходов синтаксического анализатора;
    - **Items** – информация о состояниях синтаксического анализатора;
    - **Action Map** – отображение из правил в функции, сгенерированные по атрибутам этого правила;
  - **Функции вычисления атрибутов** – файл на F#, содержащий функции для вычисления пользовательских атрибутов.
- **Ядро**, которое реализует синтаксический разбор и вычисление атрибутов и содержит:
  - **Интерпретатор таблиц**, который строит лес вывода входного выражения на основе сгенерированных таблиц. Основан на рекурсивно-восходящем алгоритме. Возвращает лес – список деревьев вывода.

– **Вычислитель атрибутов**, который обходит лес, полученный от интерпретатора таблиц, и применяет функции, найденные с помощью Action Map в сгенерированном файле, к узлам дерева.

На вход поступает файл с грамматикой, заданной пользователем и поток лексем. На выходе – результат вычислений, заданных пользователем в атрибутах грамматики.

6 Эксперименты

Был проведён ряд экспериментов, которые позволили оценить некоторые параметры инструмента.

Во всех приведённых ниже тестах грамматики описываются на языке YARD. Так же предположим, что у нас есть сторонний лексический анализатор со следующим набором лексем:

- PLUS = '+'
- MINUS = '-'
- DIV = '/'
- MULT = '*'
- LEFT = '('
- RIGHT = ')'
- NUMBER = (0..9)+
- SEMICOLON = ';'

По этому будем предполагать, что на вход инструменту поступает поток лексем.

6.1 Работа с однозначными грамматиками

Необходимо показать, что по однозначной грамматике строится инструмент имеющий линейную временную сложность. Для этого необходимо оценить количество действий LR-автомата, совершаемых при распознавании цепочки. Оно должно линейно зависеть от длинны входа.

В нашем случае операции LR-автомата – это вызовы функций *parse* и *climb*. То есть нам надо оценить зависимость количества вызовов этих функций от длинны входной цепочки.

Возьмём грамматику:

```
f : <n:string>=NUMBER {float n}
  | l=LEFT <expr:float>=e r=RIGHT {expr};
```

```
t : <l:float>=t op=(MULT {( * )} | DIV {( / )} ) <r:float>=f {op l r}
  | res=f {res};

e : res=t {res}
  | <l:float>=e op=(PLUS {( + )} | MINUS {( - )} ) <r:float>=t {op l r};

+s: res=e {res};
```

Возьмём n $=$ „2*3“ и проведём тесты для строк n'+'kn, kn $=$ $concat\ '+'$ $\underbrace{[n, n, ..., n]}_{\text{k раз}}|k = 0..99$. В результате получим график 2, где по оси Ox откладывается k, а по Oy – количество вызовов. Видно, что зависимость количества действий LR-автомата линейно зависит от длины входной цепочки.

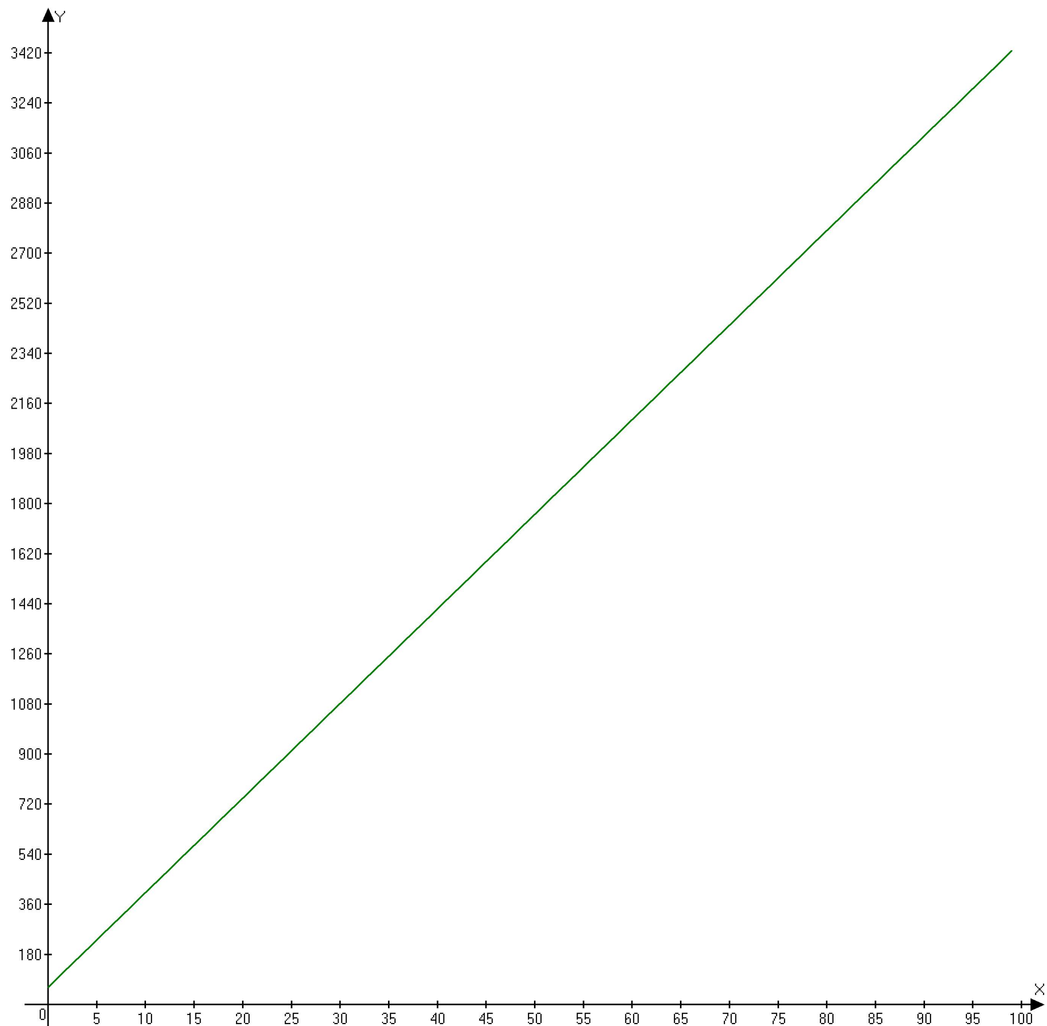


Рис. 2: Зависимость количества действий LR-автомата от длинны входной цепочки.

6.2 Возможность работы с неоднозначными грамматиками

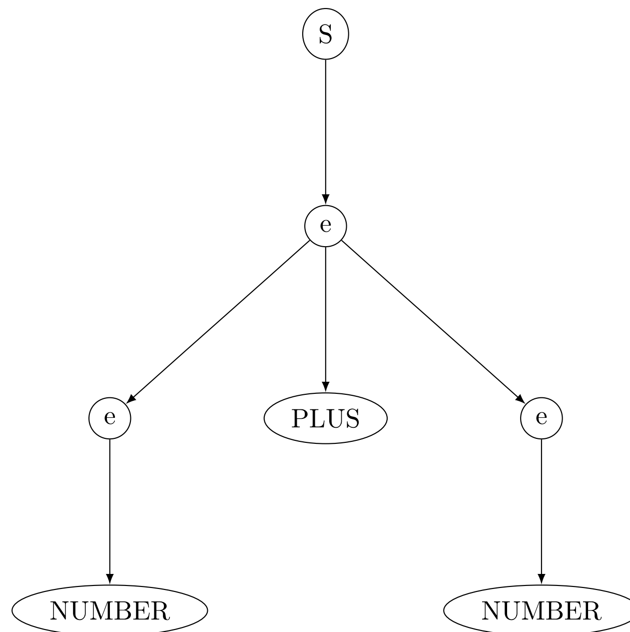
Для этого необходимо проверить, что при неоднозначной грамматике инструмент возвращает все возможные варианты вывода входной строки. В качестве примера была взята следующая грамматика:

```
+s : e;
e : e (PLUS | MINUS | MULT | DIV  ) e
  | LEFT e RIGHT
  | NUMBER ;
```

Эта грамматика описывает арифметические выражения без приоритетов. Очевидно, что данная грамматика содержит неоднозначности.

Рассмотрим несколько примеров входных цепочек:

- Пусть тестовая строка: „1+2“. Соответсвенно, список лексем: `[NUMBER; PLUS; NUMBER]`. Существует единственное дерево вывода для данной цепочки:



Инструмент так же возвращает единственное дерево:

```
<NODE name="S">
        <NODE name="e">
            <NODE name="e">
                <LEAF name="NUMBER"/>
            </NODE>
            <LEAF name="PLUS"/>
            <NODE name="e">
```

```
        <LEAF name="NUMBER"/>
      </NODE>
    </NODE>
</NODE>
```

- Строка: „1+2+3“. Список лексем: [NUMBER; PLUS; NUMBER; PLUS; NUMBER]. Для данной цепочки существует два дерева вывода и инструмент возвращает оба:

```
<NODE name="s">
    <NODE name="e">
        <NODE name="e">
            <NODE name="e">
                <LEAF name="NUMBER"/>
            </NODE>
            <LEAF name="PLUS"/>
            <NODE name="e">
                <LEAF name="NUMBER"/>
            </NODE>
        </NODE>
        <LEAF name="PLUS"/>
        <NODE name="e">
            <LEAF name="NUMBER"/>
        </NODE>
    </NODE>
</NODE>

<NODE name="s">
    <NODE name="e">
        <NODE name="e">
            <LEAF name="NUMBER"/>
        </NODE>
        <LEAF name="PLUS"/>
        <NODE name="e">
```

```
            <NODE name="e">
                <LEAF name="NUMBER"/>
            </NODE>
            <LEAF name="PLUS"/>
            <NODE name="e">
                <LEAF name="NUMBER"/>
            </NODE>
        </NODE>
    </NODE>
</NODE>
```

Таким образом, инструмент строит все возможные варианты вывода в случае, если грамматика неоднозначна.

6.3 Возможность работы с EBNF-грамматиками

Основные конструкции регулярных выражений, для которых необходимо провести проверку: последовательность, альтернатива, замыкание. Важно обратить внимание на соответствие получаемого дерева вывода ожидаемому результату. Для этого нужно проверить соответствие дерева вывода входной грамматике. В нём не должно быть новых терминалов и нетерминалов.

Для примера возьмём следующую грамматику:

```
f : NUMBER
  | LEFT e RIGHT ;

t : t (MULT | DIV ) f
  | f ;

e : t
  | e (PLUS | MINUS )t;

+s: e (SEMICOLON e )* ;
```

Данная грамматика описывает список арифметических выражений, разделённых точкой с запятой. Видно, что в ней используются все заявленные конструкции регулярных выражений (последовательность, альтернатива, замыкание).

Пусть на вход подаётся строка: „2*3; 3*4; 5*5“. Инструмент построит следующее дерево вывода:

```
<NODE name="s">
    <NODE name="e">
        <NODE name="t">
            <NODE name="t">
                <NODE name="f">
                    <LEAF name="NUMBER"/>
                </NODE>
            </NODE>
            <LEAF name="MULT"/>
            <NODE name="f">
                <LEAF name="NUMBER"/>
            </NODE>
        </NODE>
    </NODE>
    <LEAF name="SEMICOLON"/>
    <NODE name="e">
        <NODE name="t">
            <NODE name="t">
                <NODE name="f">
                    <LEAF name="NUMBER"/>
                </NODE>
            </NODE>
            <LEAF name="MULT"/>
            <NODE name="f">
                <LEAF name="NUMBER"/>
            </NODE>
        </NODE>
```

```
    </NODE>
    <LEAF name="SEMICOLON"/>
    <NODE name="e">
        <NODE name="t">
            <NODE name="t">
                <NODE name="f">
                    <LEAF name="NUMBER"/>
                </NODE>
            </NODE>
            <LEAF name="MULT"/>
            <NODE name="f">
                <LEAF name="NUMBER"/>
            </NODE>
        </NODE>
    </NODE>
</NODE>
```

Видно, что в дереве нет дополнительных нетерминалов или терминалов. Присутствуют только те, которые описаны в пользовательской грамматике, то есть инструмент реализует непосредственную поддержку EBNF-грамматик.

6.4 Поддержка s-атрибутных грамматик

Необходимо показать корректность вычисления атрибутов в случае неоднозначной грамматики. Для этого необходимо, чтобы операции с побочными эффектами работали корректно (например, проверить, что при наличии в атрибутах действия печати на экран, на экран не выводится лишней информации). Так же необходимо показать возможность вычисления атрибутов в случае расширенной контекстно-свободной грамматики.

Рассмотрим следующую грамматику:

```
+ s : t {printfn "\n reduce t rule \n"};
t : a {printfn "\n reduce a rule \n"};
t : b {printfn "\n reduce b rule \n"};
```

```
a : PLUS NUMBER {printfn "\n visit a rule \n"};
b : PLUS NUMBER SEMICOLON {printfn "\n visit b rule \n"};
```

Пусть дана строка: „+1;“ Для неё возможно две „попытки“ вывода в данной грамматике (попытка свернуть правило a и попытка свернуть правило b), но только одна из них завершится удачно. В качестве атрибутов указано действие с побочным эффектом – печать на экран. Важно убедиться, что на экран будет выведено только то, что соответствует удачному варианту разбора.

В процессе работы инструмента можно увидеть, что попытка свернуть по правилу a действительно была. Это видно, например, в трассе:

```
...
  trees:
         <NODE name="t">
            <NODE name="a">
                <LEAF name="PLUS"/>
                <LEAF name="NUMBER"/>
            </NODE>
        </NODE>
...
```

Однако на экране будет напечатано следующее:

```
 visit b rule


 reduce b rule


 reduce t rule
```

Печати, связанной с правилом a нет. Можно так же проверить дерево вывода. Инструмент получил единственное дерево:

```
<NODE name="s">
    <NODE name="t">
        <NODE name="b">
            <LEAF name="PLUS"/>
```

```
            <LEAF name="NUMBER"/>
            <LEAF name="SEMICOLON"/>
        </NODE>
    </NODE>
</NODE>
```

Таким образом, в случае, если грамматика неоднозначна, работа с действиями с побочными эффектами происходит корректно.

Теперь проверим работу вычисления атрибутов для EBNF-грамматик. Для этого рассмотрим следующую грамматику:

```
f : <n:string>=NUMBER {float n}
  | l=LEFT <expr:float>=e r=RIGHT {expr};

t : <l:float>=t op=(MULT {( * )} | DIV {( / )} ) <r:float>=f {op l r}
  | res=f {res};

e : res=t {res}
  | <l:float>=e op=(PLUS {( + )} | MINUS {( - )} ) <r:float>=t {op l r};

+s: r = e lst = (SEMICOLON  l = e {l})*
    {List.iter (printfn "result = %A \n") (r::lst)};
```

На вход подаём строку: „2*3; 3*4; 5*5“ . На выходе получаем следующий результат:

```
result = 6.0

result = 12.0

result = 25.0
```

То есть инструмент поддерживает работу с s-атрибутами при непосредственной поддержке EBNF-грамматик.

Так же, в ходе этого эксперимента было выявлено, что предложенное решение, в котором явным образом строится дерево вывода и для каждого правила строится своя семантическая функция, оказывается удобным. С

одной стороны, это позволяет упростить отладку, потому, что всегда можно проверить правильность построения дерева и в отладчике просто проконтролировать вычисление в конкретном узле (мы знаем при свёртке какого правила появился этот узел и знаем какая функция должна вычисляться). С другой – прямой доступ к лесу вывода позволяет совершать с ним дополнительные операции. Дополнительную фильтрацию или, например, печать, что оказалось полезным при получении результатов экспериментов (печать XML-представления деревьев).

7 Заключение

В ходе выполнения данной дипломной работы были получены следующие результаты:

- реализован прототип генератора синтаксических анализаторов со следующими свойствами:
  - принимает на вход s-атрибутную грамматику YARD-а и строит по ней транслятор
  - по однозначной LR-грамматике строится анализатор с линейной сложностью;
  - по неоднозначной грамматике строится анализатор, возвращающий все возможные деревья вывода для данной входной цепочки;
  - реализует поддержку EBNF-грамматик без их преобразования;
  - реализует вычисление s-атрибутов;

Кроме того, в ходе экспериментов было выяснено, что предложенный подход, при котором явно строится лес вывода, и функции вычисления атрибутов соответствуют правилам грамматики, сильно упрощает отладку целевого инструмента.

7.1 Дальнейшее развитие

Количество конструкций регулярных выражений, применяемых на практике при описании грамматики, достаточно велико, однако в инструменте поддержаны далеко не все. Ещё одним направлением развития является поддержка наследуемых атрибутов, применение которых сильно упрощает трансляцию с учётом контекста [5]. Так же необходимо реализовать поддержку предикатов (резольверов [3]). Стоит отметить, что все эти возможности реализованы на уровне входного языка инструмента YARD.

Кроме того необходимо дополнить инструмент такими ставшими привычными средствами, как автоматическое восстановление от ошибок и средства диагностики ошибок.

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] *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты. М:. Издательский дом <Вильямс>2003. 768 с.

[3] *Мартыненко Б.К.* Синтаксически управляемая обработка данных (2-е изд.). – СПб.: Издательство С.-Петербургского университета, 2004. —315 с.

[4] *Мартыненко Б.К.* Языки и трансляции. — СПб.: Издательство С.-Петербургского университета, 2004. — 229 с.

[5] *Чемоданов И.С.* Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ. 2007. —37 с.

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

[7] *Appel, Andrew W.* Compiling with continuations. // Cambridge University Press. —1992. —p. 262.
(http://portal.acm.org/citation.cfm?id=129099#)

[8] *Dick Grune, Ceriel Jacobs* Parsing techniques: a practical guide. // Ellis Horwood —1990. —p. 322.
(http://portal.acm.org/citation.cfm?id=130365&coll=GUIDE&dl=GUIDE &CFID=90477587&CFTOKEN=86670819#)

[9] *I*SO/IEC 14977 : 1996(E), "Information technology — Syntactic metalanguage — Extended BNF"

[10] *Larry Morell, David Middleton* RECURSIVE-ASCENT PARSING. // Journal of Computing Sciences in Colleges. — p. 186–201.
(http://portal.acm.org/citation.cfm?id=770849#)

[11] *Lex Augusteijn* Recursive Ascent Parsing (Re: Parsing techniques). lex@prl.philips.nl (Lex Augusteijn) Mon, 10 May 1993 07:03:39 GMT (http://compilers.iecc.com/comparch/article/93-05-045)

[12] *Mark G.J. van den Brand, Alex Sellink, Chris 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.

[13] *Leermakers René* Non-deterministic recursive ascent parsing. // Proceedings of the fifth conference on European chapter of the Association for Computational Linguistics. —1991. —p 63-68. (http://portal.acm.org/citation.cfm?id=977180.977192&coll=GUIDE&dl=GUIDE &CFID=90477587&CFTOKEN=86670819#)

[14] *Ronald Veldena* Jade, a recursive ascent LALR(1) parser generator. September 8,1998. —p. 12. (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.9503)

[15] http://apaged.mainia.de/ (сайт разработчиков APaGeD)

[16] http://dparser.sourceforge.net/ (сайт разработчиков DParser)

[17] http://dypgen.free.fr/ (сайт разработчиков Dypgen)

[18] http://parsing.codeplex.com/ (сайт разработчиков eu.h8me.Parsing)

[19] http://refactory.com/Software/SmaCC/ (сайт разработчиков SmaCC)

[20] http://semanticdesigns.com/Products/DMS/DMSToolkit.html (сайт разработчиков DMS)

[21] http://sourceforge.net/projects/gdk/ (сайт разработчиков GDK)

[22] http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/nlp/parsing/tom/0.html (сайт разработчиков Tom)

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

[24] http://www.haskell.org/ (дистрибутивы и документация по языку Haskell)

[25] http://www.haskell.org/happy/ (сайт разработчиков Happy)

[26] http://www.meta-environment.org (сайт разработчиков ASF+SDF)

[27] http://www.microsoft.com/NET/ (сайт платформы .NET)

[28] http://www.mightyheave.com/blog/?p=270 (сайт разработчиков Wormhole)

[29] http://www.research.microsoft.com/fsharp (дистрибутивы и документация по языку F#)

[30] http://www.scottmcpeak/elkhound/ (сайт разработчиков Elkhound)

[31] http://www.ultragram.com/ (сайт разработчиков UltraGram)