****

**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение   
высшего образования**

**«Московский государственный технический университет**

**Имени Н. Э. Баумана**

**(национальный исследовательский университет)»**

**(МГТУ им. Н. Э. Баумана)**

Отчёт по лабораторной работе № 4

по курсу

“Конструирование компиляторов”

Тема: “Синтаксический анализатор операторного предшествования.”

Выполнил студент группы ИУ7-21М:

Осипов Олег Константинович.

Преподаватель:

Ступников А.А.

Москва 2020 г.

**Синтаксический анализатор операторного предшествования.**

**1. Основные понятия и соглашения.**

Здесь и далее используются все соглашения, присутствующие в предыдущих лабораторных работах [3,4,5], а также те, что приняты в нижележащем тексте.

**Определение.** Операторной грамматикой G называется контекстно-свободная грамматика, которая не содержит пустых -правил, причём каждая правая часть правил такой грамматики (тела продукций) не содержат *смежные* нетерминалы.

**Пример 1.**

Ниже представлена операторная грамматика с единственным нетерминалом *E* и терминальными символами: +, -, \*, /, (, ), ^, **id**.

Стартовым символом грамматики *S* является *E*. Множество продукций *P*, состоит из следующих правил:

Данная выше грамматика не является однозначной по причине неоднозначного выбора продукции для E при разборе входной строки **id + id \* id.**

Как видно, для одной и той же входной строки существуют два различных дерева разбора. В данном случае неоднозначность проявляется в отсутствии приоритета операторов и их ассоциативности. Предположим, что операторы \*, +, - левоассоциативны а оператор возведения в степень (^) и унарный минус – правоассоциативны, и причём все они имеют следующий приоритет в порядке убывания:

|  |  |
| --- | --- |
| **Операторы** | **Приоритет** |
| (, ) | 5 |
| Унарный минус (-) | 4 |
| ^ | 3 |
| \*, / | 2 |
| +, - | 1 |

Тогда, с учётом ассоциативности и приоритета становится возможным построение синтаксического анализатора для данной грамматики, с условием, что лексический анализатор способен отличить префиксный унарный минус от бинарного (т.е. для них заданы разные токены). Для строки **id + id \* id,** будет осуществлена первая цепочка порождений, начинающихся с E + E, которая была ранее указана выше.

**2. Метод операторного предшествования.**

Метод операторного предшествования относится к классу восходящих таблично-управляемых методов синтаксического анализа на основе алгоритма типа «перенос/свертка». Этот метод основан на отношениях предшествования Вирта-Вебера, но только между терминальными символами грамматики.

**Определение.**  Для операторной грамматики G, определим три непересекающихся отношения приоритетов между терминалами грамматики и символом $, обозначающий начало и конец входной строки. Эти отношения приоритетов управляют выбором основ и имеют следующее содержание:

|  |  |
| --- | --- |
| **Отношение** | **Значение** |
|  | a уступает приоритет b (b выше a) |
|  | a и b имеют одинаковый приоритет |
|  | a забирает приоритет у b (a выше b) |

**Определение.** Для операторной грамматики G определим отношения предшествования на множестве её терминалов и символа $, игнорируя нетерминалы. Пусть *G = (N,T,P,S) – операторная грамматика,* а Тогда отношения операторного предшествования задаются следующим образом:

**Определение.** Операторная грамматика G называется *грамматикой операторного предшествования,* если между любыми двумя терминальными символами выполняется не более одного отношения предшествования.

**Пример 2**. Примером грамматики операторного предшествования служит грамматика с правилами:

Матрица отношений операторного предшествования для этой грамматики, будет иметь вид:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ( | **id** | \* | + | ) | $ |
| ) |  |  |  |  |  |  |
| **id** |  |  |  |  |  |  |
| \* |  |  |  |  |  |  |
| + |  |  |  |  |  |  |
| ( |  |  |  |  |  |  |
| $ |  |  |  |  |  |  |

Пустая клетка интерпретируется как ошибка.

Идея синтаксического анализа для грамматик операторного предшествования вытекает из теоремы, которая приводится без доказательства.

**Теорема 1.**

Пусть *G = (N,T,P,S) –* операторная грамматика и Тогда:

1) отношение операторного предшествования или выполняется между последовательными терминалами (вместе с символом $) цепочки .

2) отношение выполняется между самым правым терминалом цепочки и самым левым терминалом цепочки .

3) отношение выполняется между последовательными терминалами цепочки

4) отношение выполняется между самым правым терминалом цепочки и первым терминалом цепочки *w.*

С помощью алгоритма разбора типа «перенос-свертка» легко выделить терминальные символы, входящие в основу . Однако возникают проблемы в связи с нетерминальными символами, поскольку на них не определены отношения операторного предшествования. Тем не менее, тот факт, исходная грамматика является операторной грамматикой, позволяет строить «остовный» правый разбор.

**Определение.** Пусть *G = (N,T,P,S) – операторная грамматика.* Остовной грамматикой для G назовём грамматику содержащую каждое правило для которого в *P* найдется такое правило что:

1)

2)

Следует обратить внимание на то, что и, вообще говоря, может содержать цепочки, не принадлежащие .

**Пример 3.**

Для операторной грамматики G из примера 2, соответствующая остовная граматика будет содержать следующее множество правил :

Теперь можно описать алгоритм типа «перенос-свертка» для грамматик операторного предшествования.

**2.1. Алгоритм построение анализатора операторного предшествования.**

*Вход: G = (N,T,P,S) –* грамматика операторного предшествования.

*Выход:* Алгоритм разбора *A = (f, g)* типа “перенос-свёртка” для остовной грамматики .

*Метод:* Пусть означает S или , а *shift* и *r* – перенос и свёртку.

1)

2)

3)

4)

5)

a)

b)

c) отношение = выполняется между последовательными терминальными символами цепочки , если они существуют

d)

6)

**2.2 Эвристический метод нахождения отношений операторного предшествования.**

Для грамматик математических выражений, наподобие грамматики из примера 3, можно использовать следующий эвристический метод для создания корректного множества отношений операторного предшествования.

1) Если оператор op1 имеет более высокий приоритет, чем оператор op2, определим op1 op2 и op2 op1. Например, приоритет \* > приоритета +, то \* + и + \*. Эти отношения гарантируют, что при получении выражений вида E + E \* E + E, основой является E \* E и именно она будет свёрнута первой.

2) Если операторы op1 и op2 представляют собой операторы равного приоритета (это может быть один и тот же оператор), то устанавливаем, что

op1 op2 и op2 op2 если операторы лево-ассоциативны и op1 op2 op2 op1 в случае, если операторы право-ассоциативны. Например, для левых ассоциативных операторов + и – устанавливаем: + - и - +. Для право-ассоциативного оператора возведения в степень (его можно определить в грамматике из примера 3 как оператор с наивысшим приоритетом) следует принять ^ ^. Такие отношения гарантируют, что в выражениях вида E – E + E основой является E – E, а в выражении E ^ E ^ E – основой является последнее подвыражение E ^ E.

3) Для всех операторов op определяем следующие отношения:

op **id,** op (, ( op, $ op, **id**  op, op ), ) op, op

Кроме того, считаем, что ( = ), $ (, $ **id**, ( (, **id** $, ) $, ( **id**,

) ), **id** ). Эти правила гарантируют, что **id,** и ( E ) будут свёрнуты к E. Кроме того, $ служит маркером как левого так и правого конца строки, что заставляет основы находится между ними.

4) Если грамматика содержит унарный префиксный оператор (например, логическое отрицание ~ для логических выражений), который не имеет бинарного аналога, то op ~ для любого оператора op, независимо от того, является ли он бинарным или унарным. Кроме того, ~ op, если ~ имеет более высокий приоритет, чем op, и ~ op в противном случае. Например, если отрицание ~ имеет более высокий приоритет, чем конъюнкция &, и & - лево-ассоциативный оператор, то в соответствии с приведёнными правилами, выражение E & ~E & E должно быть сгруппировано как (E & (~E)) & E. Правила для постфиксного унарного оператора аналогичны рассмотренным правилам. Ситуация изменяется при рассмотрении операторов типа знака «минус», который может использоваться и как унарный префиксный оператор, и как бинарный инфиксный. Наилучшим решением этой проблемы было бы распознавание типа оператора лексическим анализатором и использование им разных токенов для бинарного и унарного минусов. К сожалению, лексическому анализатору для этого придётся хранить предыдущий прочитанную лексему для определения типа токена для минуса.

**2.3 Обработка и нейтрализация ошибок при синтаксическом анализе операторного предшествования.**

Для каждой пустой клетки матрицы отношений операторного предшествования необходимо определить подпрограмму обработки и нейтрализации ошибок; одна и та же подпрограмма может использоваться в нескольких местах. Для матрицы отношений операторного предшествования из примера 2 можно предложить следующие подпрограммы.

|  |
| --- |
| Подпрограмма el  Причина ошибки: отсутствует выражение в целом.  Действие анализатора: вставить операнд во входной поток.  Сообщение об ошибке: пропущен операнд. |
| Подпрограмма e2  Причина ошибки: выражение начинается с правой скобки.  Действие анализатора: удалить правую скобку из входного потока.  Сообщение об ошибке: несбалансированная правая скобка. |
| Подпрограмма e3  Причина ошибки: операнд или правая скобка следует за операндом или левой скобкой.  Действие анализатора: вставит знак операции (например, +) во входной поток.  Сообщение об ошибке: пропущен оператор. |
| Подпрограмма *e*4  Причина ошибки: выражение завершается левой скобкой.  Действие анализатора: вытолкнуть левую скобку из стека.  Сообщение об ошибке: пропущена правая скобка. |

При таком механизме обработки и нейтрализации ошибок матрица отношений операторного предшествования из примера 2 примет следующий вид:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ( | **id** | \* | + | ) | $ |
| ) | e3 | e3 |  |  |  |  |
| **id** | e3 | e3 |  |  |  |  |
| \* |  |  |  |  |  |  |
| + |  |  |  |  |  |  |
| ( |  |  |  |  | = | e4 |
| $ |  |  |  |  | e2 | e1 |

Возможны и другие ошибки, которые возникают тогда, когда основа обнаружена, но не существует правила вывода с правой частью, соответствующей основе. Обработка таких ошибок требует творческого подхода.

**2.4 Множества L, R, Lt, Rt для нетерминалов грамматики.**

Для каждого нетерминального символа U, и каждого терминального символа t определим следующие множества.

Множество крайних левых символов для нетерминала U, это множество *L(U),* такое, что

Множество крайних правых символов для нетерминала U, это множество *R(U),* такое, что

Здесь, символ C – любой грамматический символ грамматики (терминал или нетерминал), а z – произвольная цепочка символов грамматики.

Множества крайних левых и крайних правых терминальных символов определяются следующим образом:

- множество крайних левых терминальных символов, относительно нетерминального символа U.

- множество крайних левых терминальных символов, относительно нетерминального символа U. Здесь, C – нетерминальный символ, z – произвольная цепочка грамматических символов, а t – терминальный символ грамматики.

Множества *L(U), R(U)* могут быть построены для каждого нетерминального символа грамматики, очень просто. Просматриваем все продукции, заголовок которых имеет имя текущего нетерминального символа. Добавляем первый грамматический символ из тела продукции во множество L, и последний грамматический символ во множество R. В случае, если тело продукции содержит одиночный символ, то он добавляется в оба множества. Если символ является терминальным, то переходим к следующей продукции для текущего нетерминала (т.е. к следующей альтернативе). Иначе если это нетерминальный символ и он имеет имя, отличное от имени заголовка продукции, то перейти к рассмотрению его продукций. При рассмотрении текущей продукции выполняем тоже самое: добавляем крайний левый символ и крайний правый символ продукции во множества L и R, если его там нет. Если это снова нетерминальный символ, и он также имеет другое имя, отличное от имени рассматриваемой продукции, то также рассматриваем его правила. Т.е. мы рекурсивно проверяем для каждого нетерминального символа, который появляется слева или справа его продукции, до тех пор, пока не встретим терминал. Встретив терминал, мы выходим из рекурсии, и переходим к просмотру следующей альтернативы для текущего нетерминального символа U. После просмотра всех альтернатив, мы окончательно получим множества *L(U), R(U)* для нетерминала U.

Выше описанная процедура очень похожа на построение множества

*FIRST(z)* для строки грамматических символов грамматики. Отличие заключается в том, что для нетерминала, которого мы добавляем, мы просматриваем не то, что находится справа от него в текущей продукции, а то, что он порождает (т.е. его продукции) и в расчёт берётся не то, что его множество FIRST содержит символ пустой строки, а то, что данный символ является нетерминалом.

Следует отметить, что данные множества строятся для операторной грамматики, которые по определению не содержат пустых правил, и, следовательно не содержат символа пустой строки .

Множества крайних левых и крайних правых терминальных символов для нетерминала U, (, ) могут быть построены следующим образом.

1. В первую очередь для нетерминала U необходимо вычислить множества *L(U), R(U).*

2. Для нетерминала U просматриваются правила, вида Крайние левые терминальные символы включаются во множество . Аналогично для ищутся правила вида (т.е. во множество записываются все крайние слева терминальные символы из правых частей правил для U, а в – все крайние справа терминальные символы этих правил). Не исключено, что один и тот же терминальный символ попадёт в оба множества.

3. Если множество *L(U)* не содержит никаких других нетерминальных символов, отличных от U, то построение завершено. Иначе, мы должны дополнить множество всеми новыми символами (не входящими в из множеств Аналогичная операция проводится выполняется для множества на основе множества *R(U).*

**Пример 4.**

Рассмотрим операторную грамматику из примера 2. Вычислим для нетерминалов E,T,F их соответствующие множества L,R, Lt, Rt.

Нетерминал E имеет две продукции Первая альтернатива добавляет в L сам нетерминал E, а в R – нетерминал T. Поскольку T отличается от заголовка продукции, то мы должны рассмотреть его продукции, для продолжения построения множества R. Нетерминал T имеет следующие продукции: . Во множество R добавляется новый нетерминал F, который отличается от заголовка продукции T. Рассмотрим также его продукции: **.** Просмотр добавит в R два терминальных символа ), и **id**. Так как больше нетерминалов нет, мы переходим к следующей альтернативе для E. Следующее правило для E добавит во множество L новый нетерминал T. Множество R уже содержит данный нетерминал. Поскольку T отличается от заголовка продукции, то выполняем те же шаги, что были сделаны, но только теперь мы строим множество L и будем выбирать крайние слева символы. В итоге, множества L(E) и R(E) будут равны:

**id** }. **id** }.

Ниже дана таблица множеств L,R для данной грамматики.

|  |  |  |
| --- | --- | --- |
| Нетерминал | L(U) | R(U) |
| E | E,T,F,(,**id** | T,F,),**id** |
| T | T,F,(,**id** | F,),**id** |
| F | (,**id** | ),**id** |

Для нетерминала F множества Lt и Rt равны соответствующим множествам L(F) и R(F). Для T во множество Lt и Rt добавится терминальный символ \*. Так как множества L(T) и R(T) содержат нетерминальный символ F, отличный от T, то мы также должны добавить во множества Lt и Rt множества Lt(F) и Rt(F), полученные на предыдущем шаге.

Ниже дана таблица множеств Lt, Rt для нетерминалов данной грамматики:

|  |  |  |
| --- | --- | --- |
| Нетерминал | Lt(U) | Rt(U) |
| E | +,\*,(,**id** | +,\*,),**id** |
| T | \*,(,**id** | \*,),**id** |
| F | (,**id** | ),**id** |

Используя построенные множества Lt и Rt для нетерминалов грамматики, можно построить матрицу операторного предшествования, эквивалентную из примера 2.

**2.5 Алгоритм построения матрицы операторного предшествования на основе множеств Lt, Rt.**

*Вход:* Множества операторной грамматики G.

*Выход:* Матрица операторного предшествования M для любой пары терминальных символов грамматики.

*Метод:*

1. Для символа $, обозначающего начало и конец входной строки, и терминальными символами ставим следующее отношение:

a) ставим знак на пересечении строки, обозначающей $ и столбца, обозначающего терминал a, для каждого терминала из множества

b) ставим знак на пересечении строк, обозначающих терминальный символ из множества и столбца, обозначающего символ $.

2. Берём первый символ из множества терминальных символов грамматики T. Положим i = 1.

3. Во всём множестве правил **P**, ищем правила вида: , где – текущий терминальный символ, – произвольный терминальный символ, U, C – одиночные нетерминальные символы, а x,y – произвольные строки грамматических символов. Фактически, происходит поиск правил, в которых символы либо стоят рядом друг за другом, либо между ними стоит ровно один нетерминал, причём стоит слева от .

4. Для всех символов , найденных на предыдущем шаге, ставим в клетке матрицы, на пересечении строки и столбца знак =.

5. Во всём множестве правил **P**, ищем правила вида: , где – текущий терминальный символ, C – одиночные нетерминальные символы, а x,y – произвольные строки грамматических символов. Фактически, происходит поиск правил, в которых символ стоит слева от нетерминала .

6. Для всех найденных нетерминалов на предыдущем шаге берём множество Для всех терминальных символов , входящих в это множество делаем следующее, ставим знак на пересечении строки, помеченной символом и столбцом, помеченного символом

7. Во всём множестве правил **P,** ищем правила вида: где – текущий терминальный символ, C – одиночные нетерминальные символы, а x,y – произвольные строки грамматических символов. Фактически, происходит поиск правил, в которых символ стоит справа от нетерминала .

8. Для всех найденных нетерминалов на предыдущем шаге берём множество Для всех терминальных символов , входящих в это множество делаем следующее, ставим знак на пересечении строки, помеченной символом и столбцом, помеченного символом .

Если при построении матрицы на любом шаге не возникало противоречий, когда в одну и ту же клетку надо было записать два или три отношения, то матрица построена правильно. Если на каком-то шаге возникло противоречие, то данная грамматика G, не является грамматикой операторного предшествования.

**3. Постановка задачи.**

Рассматривается дополненная грамматика из лабораторной работы № 3. В данной грамматике были добавлены правила для булевых (логических) выражений, а также изменено правило для унарного минуса. Отметим, что грамматика из лабораторной работы № 3 не является операторной, по причине наличии смежных нетерминалов в правых частях некоторых правил. Поэтому данная грамматика была модифицирована, для того, чтобы она удовлетворяла определению операторной грамматики. В частности, из неё были удалены правила, представляющие бинарные операторы: RELOP, PLUSOP, MULLOP. Множество правил модифицированной грамматики даны ниже. Заголовок первого правила является стартовым символом грамматики. Для грамматических символов и правил используются соглашения, принятые из лабораторной работы № 2. Поскольку символы |, используются как мета-символы описания грамматики, то для того, чтобы подчеркнуть, что данные символы являются частью правил (т.е. они грамматические символы), используются двойные кавычки.

| **not** ( IMPL ) | **realNum | true | false**

Здесь терминалы **um, up** обозначают унарный минус и плюс соответственно. Для данной грамматики необходимо разработать программу восходящего синтаксического анализа, используя алгоритм “сдвиг-свёртка” для грамматик операторного предшествования.

**3. Реализация программы.**

Модифицируем разработанную программу из предыдущей лабораторной работы. Данная программа умеет уже генерировать нисходящий синтаксический анализатор вместе с лексическим. Необходимо добавить генерацию восходящего синтаксического анализа, использующего алгоритм “сдвиг-свёртка” для грамматик операторного предшествования. В первую очередь, необходимо устранить неоднозначность между терминалами для бинарного минуса, плюса и их унарных аналогов. Лексический анализатор должен знать, когда возвращать бинарный оператор, а когда унарный для одной и той же лексемы. Для этого анализатор должен знать имя предыдушей лексемы. Если положить, что предыдущая лексема была операндом, то логично, что лексема для минуса и плюса, это лексема бинарного оператора. Если предыдущая лексема не была операндом, то текущий минус или плюс являются унарными операторами. Для литералов “true”, “false” лексический анализатор возвращает не идентификатор, а саму лексему (имя лексемы совпадает со значением). Для этого были определены ключевые слова и изменён входной формат грамматики. Теперь, помимо предыдущих 4 секции, можно определить секцию “keywords”, содержащую список ключевых слов. (В json-е списком является массив). Ключевые слова представлены строками. Для информации об операндах и унарных операторов также можно определить секцию “meta”. Секция “meta” представляет из себя Json-объект и состоит из двух свойств “operands” и “aliases”. Первая является списком операндов (формат аналогичный ”keywords”). Вторая секция также является объектом. Она определяет список пар имя:значение, где имя – это имя лексемы для бинарного оператора, а значение – имя лексемы для унарного оператора. В случае, если минус является бинарным оператором, то возвращается лексема “-“, иначе будет возвращена имя лексемы для унарного минуса – “um”.

Поскольку символы |,\*,+,[,] – это мета-символы (т.е. операторы) регулярных выражений, то для того, чтобы их использовать как обычные символы, необходимо поставить перед ними символ собаки @. Соответственно для использования символа собаки необходимо его продублировать - @@. Если регулярное выражение состоит ровно из одного символа, то “@” использовать не надо. В этом случае символ не будет считаться оператором регулярного выражения.

С учётом сделанных предположений, итоговый входной файл грамматики выглядит следующим образом:

{  
 **"terms"**: {  
 **"+"**: **"+"**,  
 **"-"**: **"-"**,  
 **"\*"**: **"\*"**,  
 **"/"**: **"/"**,  
 **"%"**: **"%"**,  
 **"^"**: **"^"**,  
 **"("**: **"("**,  
 **")"**: **")"**,  
 **"{"**: **"{"**,  
 **"}"**: **"}"**,  
 **"<"**: **"<"**,  
 **"<="**: **"<="**,  
 **"="**: **"="**,  
 **"=="**: **"=="**,  
 **">"**: **">"**,  
 **">="**: **">="**,  
 **"<>"**: **"<>"**,  
 **";"**: **";"**,  
 **"&&"** : **"&&"**,  
 **"||"** : **"@|@|"**,  
 **"~"** : **"~"**,  
 **"->"**: **"->"**,

**"."**: **"."**,  
 **"id"**: **"[A-Za-z\_]([A-Za-z\_0-9]+|empty)"**,  
 **"num"**: **"[0-9]+"**,  
 **"realNum"**: **"[0-9]+.[0-9]+(((E|e)(-|empty)[0-9]+)|empty)"**,  
 **"empty"**: **null** },  
 **"keywords"**: [**"true"**,**"false"**],  
 **"meta"**: {  
 **"operands"**: [**"id"**,**"num"**,**"realNum"**,**"true"**,**"false"**],  
 **"aliases"**: {  
 **"-"**: **"um"**,  
 **"+"**: **"up"** }  
 },  
  
 **"nonTerms"**: [**"E"**,**"AE"**,**"T"**,**"F"**,**"PE"**,**"B"**,**"OP"**,**"OPLIST"**,**"P"**,**"IMPL"**,**"CONJ"**,**"DISJ"**,**"UE"**],  
 **"productions"**: [{**"P"**: [**"{"**,**"OPLIST"**,**"}"**,**"."**]},  
 {**"OPLIST"**: [**"OPLIST"**,**";"**,**"OP"**]},{**"OPLIST"**: **"OP"**},  
 {**"OP"**: [**"id"**,**"="**,**"IMPL"**]},{**"OP"**: [**"{"**,**"OPLIST"**,**"}"**]},  
 {**"IMPL"**: [**"IMPL"**,**"->"**,**"DISJ"**]},{**"IMPL"**: **"DISJ"**},  
 {**"DISJ"**: [**"DISJ"**,**"||"**,**"CONJ"**]},{**"DISJ"**: **"CONJ"**},  
 {**"CONJ"**: [**"CONJ"**,**"&&"**,**"E"**]},{**"CONJ"**: **"E"**},  
 {**"E"** : [**"AE"**,**"<"**,**"AE"**]},{**"E"** : [**"AE"**,**"<="**,**"AE"**]},{**"E"** : [**"AE"**,**">"**,**"AE"**]}  
 ,{**"E"** : [**"AE"**,**">="**,**"AE"**]},{**"E"** : [**"AE"**,**"=="**,**"AE"**]},{**"E"** : [**"AE"**,**"<>"**,**"AE"**]},{**"E"**: **"AE"**}  
 ,{**"AE"**: [**"T"**]},{**"AE"**: [**"AE"**,**"+"**,**"T"**]},{**"AE"**: [**"AE"**,**"-"**,**"T"**]}  
 ,{**"T"** : [**"F"**]}, {**"T"**: [**"T"**,**"\*"**,**"F"**]},{**"T"**: [**"T"**,**"/"**,**"F"**]},{**"T"**: [**"T"**,**"%"**,**"F"**]}  
 ,{**"F"**: [**"UE"**]},{**"F"**: [**"F"**,**"^"**,**"UE"**]}  
 ,{**"UE"**: [**"~"**,**"UE"**]},{**"UE"**: [**"up"**,**"UE"**]},{**"UE"**: [**"um"**,**"UE"**]},{**"UE"**: **"PE"**}  
 ,{**"PE"**: **"num"**},{**"PE"**: **"id"**},{**"PE"**: **"realNum"**},{**"PE"**: **"true"**},{**"PE"**: **"false"**},{**"PE"**: [**"("**,**"IMPL"**,**")"**]}  
 ],  
 **"start"**: **"P"**}

Листинг 1. Входной файл грамматики.

Ниже дан код, модифицируемый конструктор класса Grammar для распознования новых секций.

**public class** Grammar {

…

**private** Set<String> **operands**;  
**private** Map<String,String> **aliases**;

**public** Grammar(Set<String> T, Set<String> N, Map<String,Set<GrammarString>> P,String start, String em, Map<String,List<String>> lexs, Set<String> kws, Set<String> ops, Map<String,String> als){  
 **this**.**T** = T;  
 **this**.**N** = N;  
 **this**.**P** = P;  
 **this**.**S** = start;  
 **this**.**E** = em;  
 **this**.**lex\_rules** = lexs;  
 **this**.**keywords** = kws;  
 **this**.**operands** = ops;  
 **this**.**aliases** = als;  
 computeN\_g();  
 computeN\_e();  
}  
  
**public** Grammar(JsonObject jsonG){  
 **this**.**keywords** = **new** HashSet<>();  
 **this**.**operands** = **new** HashSet<>();  
 **this**.**aliases** = **new** HashMap<>();

… //Compute terms and non-terms.

*//optional keywords.*JsonElement K = jsonG.getElement(**"keywords"**);  
**if**(K **instanceof** JsonArray){  
 ArrayList<JsonElement> kws = ((JsonArray) K).getElements();  
 **for**(JsonElement e : kws){  
 **if**(e **instanceof** JsonString){  
 **this**.**keywords**.add(((JsonString) e).getValue());  
 **this**.**T**.add(((JsonString) e).getValue());  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Expected String value of reserved keyword."**,**null**);  
 }  
}  
  
*//optional meta*JsonElement M = jsonG.getElement(**"meta"**);  
**if**(M **instanceof** JsonObject){  
 JsonObject o = (JsonObject) M;  
 JsonElement opd = o.getElement(**"operands"**);  
 **if**(opd **instanceof** JsonArray){  
 ArrayList<JsonElement> kws = ((JsonArray) opd).getElements();  
 **for**(JsonElement e : kws){  
 **if**(e **instanceof** JsonString && (**this**.**T**.contains(((JsonString) e).getValue()) || **this**.**keywords**.contains(((JsonString) e).getValue()) ) ){  
 **this**.**operands**.add(((JsonString) e).getValue());  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Expected String being contained in terms or keywords."**,**null**);  
 }  
 }  
 JsonElement als = o.getElement(**"aliases"**);  
 **if**(als **instanceof** JsonObject){  
 JsonObject al = (JsonObject) als;  
 Set<String> termNames = al.getValue().keySet();  
 **for**(String t : termNames){  
 **if**(!**this**.**T**.contains(t) && !**this**.**keywords**.contains(t))  
 **throw new** InvalidJsonGrammarException(**"Expected String being contained in terms or keywords."**,**null**);  
 JsonElement el = al.getElement(t);  
 **if**(el **instanceof** JsonString) {  
 String val = ((JsonString) el).getValue();

**this**.**T**.add(val);

**this**.**aliases**.put(t,val);

}

**else  
 throw new** InvalidJsonGrammarException(**"Expected String value."**,**null**);

}  
 }  
}

…

… //continue computing of productions.

}

**public** Set<String> getKeywords(){ **return this**.**keywords**; }  
  
**public** Set<String> getOperands(){ **return this**.**operands**;}  
  
**public** Map<String,String> getAliases(){ **return this**.**aliases**;}

Листинг 2. Чтение новых данных из секций “keywords”, “meta”.

Вычисление данных из секций “keywords” и “meta” выполняется после вычисления множества терминалов. Секция “meta” читается после секции “keywords”. Ниже даны код изменённых процедур *addConcat* и *buildNFA* из лаб. работы № 1, для обработки символа “@”.

*//Algorithm: Mac Naughton-Yamada-Tompson (Мак-Нотона, Ямады, Томпсона)***public static** CNFA buildNFA(LinkedStack<Character> expr, RegexRPNParser parser, Elem<Integer> el){  
 LinkedStack<CNFA> result = **new** LinkedStack<>();  
 HashSet<Character> alpha = **new** HashSet<>();  
 **int** c = el.getV1();  
 **int** pos = 0;  
 Iterator<Character> itr = expr.iterator();  
 **while**(itr.hasNext()){  
 **char** tok = itr.next();  
 **if**(parser.isUnaryOp(tok)){  
 CNFA g = result.top();  
 result.pop();  
 **for**(Vertex v: g.getNodes()){*//nullify finish* v.setFinish(**false**);  
 }  
 Vertex s = **new** Vertex(c+**""**);  
 c++;  
 Vertex t = **new** Vertex(c+**""**);  
 c++;  
 Edge iloop = **new** Edge(g.getFinish(), g.getStart(), (**char**) 1);  
 g.getFinish().setFinish(**false**);  
 g.getStart().setStart(**false**);  
 Edge se = **new** Edge(s, g.getStart(), (**char**) 1);  
 Edge fe = **new** Edge(g.getFinish(), t, (**char**) 1);  
 **if**(tok == **'\*'**) {*// '+' and '\*' differ only with one edge.* Edge loop = **new** Edge(s, t, (**char**) 1);*//for '\*' add empty from start to finish* }  
 CNFA R = **new** CNFA();  
 t.setFinish(**true**);  
 R.setStart(s);  
 R.setFinish(t);  
 result.push(R);  
 pos++;  
 }  
 **else if**(parser.isOperator(tok)){  
 CNFA g2 = result.top();  
 result.pop();  
 CNFA g1 = result.top();  
 result.pop();  
 **for**(Vertex v: g2.getNodes()){*//nullify finish* v.setFinish(**false**);  
 }  
 **for**(Vertex v: g1.getNodes()){  
 v.setFinish(**false**);  
 }  
 **if**(tok == **'^'**) {  
 Vertex inter = g1.getFinish();  
 List<Edge> outE = g2.getStart().getEdges().stream().filter(edge -> edge.getSource().equals(g2.getStart())).collect(Collectors.*toList*());  
 List<Edge> outEi = g2.getStart().getEdges().stream().filter(edge -> edge.getTarget().equals(g2.getStart())).collect(Collectors.*toList*());  
 inter.setFinish(**false**);  
 **for**(Edge e: outE){*//union output edges.* Edge ae = **new** Edge(inter,e.getTarget(),e.getTag());  
 g2.disconnectVertexByEdge(e,g2.getStart(),e.getTarget());  
 }  
 **for**(Edge e : outEi){*//union input edges.* Edge ea = **new** Edge(e.getSource(),inter,e.getTag());  
 g2.disconnectVertexByEdge(e,e.getSource(),g2.getStart());  
 }  
 CNFA FC = **new** CNFA();  
 g2.getFinish().setFinish(**true**);  
 FC.setStart(g1.getStart());  
 FC.setFinish(g2.getFinish());  
 result.push(FC);  
 pos++;  
 }  
 **else if**(tok == **'|'**){  
 Vertex s = **new** Vertex(c+**""**);  
 c++;  
 Vertex t = **new** Vertex(c+**""**);  
 c++;  
 Vertex s1 = g1.getStart();  
 Vertex s2 = g2.getStart();  
 Vertex t1 = g1.getFinish();  
 Vertex t2 = g2.getFinish();  
 Edge s\_s1 = **new** Edge(s,s1,(**char**)1);  
 Edge s\_s2 = **new** Edge(s,s2,(**char**)1);  
 Edge t\_t1 = **new** Edge(t1,t,(**char**)1);  
 Edge t\_t2 = **new** Edge(t2,t,(**char**)1);  
 s.setStart(**true**);  
 s1.setStart(**false**);  
 s2.setStart(**false**);  
 t1.setFinish(**false**);  
 t2.setFinish(**false**);  
 t.setFinish(**true**);  
 CNFA FU = **new** CNFA();  
 FU.setStart(s);  
 FU.setFinish(t);  
 result.push(FU);  
 pos++;  
 }  
 }  
 **else**{*//token is not operator.* Vertex v1 = **new** Vertex(c+**""**);  
 c++;  
 Vertex v2 = **new** Vertex(c+**""**);  
 c++;  
 v2.setFinish(**true**);  
 **if**(tok == **'@'**) {  
 *//System.out.println(tok);* tok = itr.next();*//get operand after @ symbol.  
 //System.out.println(tok);* }  
 Edge e = **new** Edge(v1,v2,tok);  
 CNFA F = **new** CNFA();  
 F.setComboStart(v1);  
 F.setFinish(v2);  
 alpha.add(tok);  
 result.push(F);  
 pos++;  
 }  
 }  
 result.top().setAlpha(alpha);  
 el.setV1(c);  
 **return** result.top();  
}  
  
**public static** String addConcat(String s,RegexRPNParser parser){  
 StringBuilder result = **new** StringBuilder();  
 **for**(**int** i = 0; i < s.length(); i++){  
 **if**(s.charAt(i) == **'['**){*// replace class [A-Z] with expression (A|B|...|Z) and add '^' if needed.* **char** t = s.charAt(i);  
 **if**(i > 0 && s.charAt(i - 1) != **'('** && s.charAt(i - 1) != **'|'**){  
 result.append(**'^'**);  
 }  
 result.append(**'('**);  
 **int** j = i + 1;  
 **boolean** wflag = **false**;  
 **while**(t != **']'** && j < s.length()){  
 t = s.charAt(j);  
 **if**(t == **'@'**){  
 result.append(**'@'**);  
 result.append(s.charAt(j + 1));  
 **if**(j + 2 < s.length() && s.charAt(j + 2) != **']'**)  
 result.append(**"|"**);  
 j++;  
 j++;  
 **continue**;  
 }  
 **if**(j + 1 < s.length() && s.charAt(j) == **'-'**){  
 **char** a = s.charAt(j - 1);  
 **char** b = s.charAt(j + 1);  
 **if**(a > b){  
 **char** temp = a;  
 a = b;  
 b = temp;  
 }  
 **while**(a != b){  
 a++;  
 result.append(a).append(**"|"**);  
 }  
 CharSequence subor = result.subSequence(0,result.length() - 1);  
 result = **new** StringBuilder().append(subor);  
 j = j + 2;  
 wflag = **true**;  
 **continue**;  
 }  
 **if**(wflag && t != **']'**){  
 result.append(**"|"**);  
 wflag = **false**;  
 }  
 String or = (j + 1 == s.length() || s.charAt(j + 1) == **']'**) ? s.charAt(j)+**""** : s.charAt(j)+**"|"**;  
 result.append(s.charAt(j) == **']'** ? **")"** : or);  
 **if**(s.charAt(j) == **']'**){  
 j++;  
 **break**;  
 }  
 **if**(s.charAt(j + 1) == **']'**){  
 result.append(**')'**);  
 j++;  
 j++;  
 **break**;  
 }  
 j++;  
 }  
 i = j;  
 **if**(i == s.length())  
 **return** result.toString();  
 *//check after [] current symbol.* **if**((s.charAt(i) == **')'** || s.charAt(i) == **'\*'** || s.charAt(i) == **'+'**)){  
 result.append(s.charAt(i));  
 **if**(i + 1 < s.length() && s.charAt(i + 1) ==**'@'**) {  
 result.append(s.charAt(i + 1));  
 **if**(i + 2 < s.length()) {  
 result.append(s.charAt(i + 2));  
 i++;  
 }  
 i++;  
 }  
 **else if**(i + 1 < s.length() && s.charAt(i + 1) != **')'** && s.charAt(i + 1) != **'+'** && s.charAt(i + 1) != **'\*'** && s.charAt(i + 1) != **'|'** && s.charAt(i + 1) != **'['**)  
 result.append(**'^'**);  
 }  
 **else if**(s.charAt(i) == **'@'**){  
 result.append(**'@'**);  
 **if**(i + 1 < s.length())  
 result.append(s.charAt(i + 1));  
 i++;  
 }  
 **else if**(parser.isTerminal(s.charAt(i)) || s.charAt(i) == **'('**){  
 result.append(**'^'**).append(s.charAt(i));  
 }  
 **else if**(s.charAt(i) == **'['**)  
 i = j - 1;  
 **continue**;  
 }  
 result.append(s.charAt(i));  
 **if**(s.charAt(i) == **'@'**){  
 **if**(i + 1 < s.length()) {  
 result.append(s.charAt(i + 1));  
 **if**(i + 2 < s.length() && (parser.isTerminal(s.charAt(i + 2)) || s.charAt(i + 2) == **'('** ))  
 result.append(**'^'**);  
 i++;  
 **continue**;  
 }  
 }  
 **if**(parser.isTerminal(s.charAt(i)) && i + 1 < s.length() && (parser.isTerminal(s.charAt(i + 1)) || s.charAt(i + 1) == **'('** ) ){  
 result.append(**'^'**);  
 }  
 **if**((s.charAt(i) == **')'** || s.charAt(i) == **'\*'** || s.charAt(i) == **'+'** ) && i + 1 < s.length() && (parser.isTerminal(s.charAt(i + 1)) || s.charAt(i + 1) == **'('**) ){  
 result.append(**'^'**);  
 }  
 }  
 **return** result.toString();  
}

Листинг 3. Построение автомата по регулярному выражению с учётом обработки символа @.

Теперь, изменим тип DFALexer, чтобы он обрабатывал ключевые слова и унарные операторы.

**package** ru.osipov.labs.lab3.lexers;  
  
**import** java.io.IOException;  
**import** java.io.InputStream;  
**import** java.util.Map;  
**import** java.util.Set;  
  
**public interface** ILexer {  
 Token recognize(InputStream f) **throws** IOException;  
 Token generateError(String s1, String s2);  
 **void** reset();  
 **void** setKeywords(Set<String> s);  
 Set<String> getKeywords();  
 **void** setOperands(Set<String> ops);  
 Set<String> getOperands();  
 **void** setAliases(Map<String,String> aliases);  
 Map<String,String> getAliases();  
}

**package** ru.osipov.labs.lab3.lexers;  
  
**import** ru.osipov.labs.lab1.structures.automats.CNFA;  
**import** ru.osipov.labs.lab1.structures.automats.DFA;  
**import** ru.osipov.labs.lab1.structures.automats.NFA;  
**import** ru.osipov.labs.lab1.structures.graphs.Pair;  
**import** ru.osipov.labs.lab1.structures.graphs.Vertex;  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
  
**import** java.io.IOException;  
**import** java.io.InputStream;  
**import** java.util.\*;  
**import** java.util.stream.Collectors;  
  
**public class** DFALexer **extends** DFA **implements** ILexer {  
  
 **private** LexerIO **io**;  
 **private** Set<String> **keywords**;  
 **private** Set<String> **operands**;  
 **private** Map<String,String> **aliases**;  
 **private** Token **prevTok**;  
  
 **public** DFALexer(DFA dfa, LexerIO io){  
 **super**(dfa,**true**);  
 **this**.deleteDeadState();  
 **this**.**io** = io;  
 **this**.**keywords** = **new** HashSet<>();  
 **this**.**operands** = **new** HashSet<>();  
 **this**.**aliases** = **new** HashMap<>();  
 **this**.**prevTok** = **null**;  
 }  
  
 **public** DFALexer(NFA nfa, LexerIO io){  
 **super**(nfa);  
 **this**.deleteDeadState();  
 **this**.**io** = io;  
 **this**.**keywords** = **new** HashSet<>();  
 **this**.**operands** = **new** HashSet<>();  
 **this**.**aliases** = **new** HashMap<>();  
 **this**.**prevTok** = **null**;  
 }  
  
 **public** DFALexer(CNFA nfa){  
 **super**(nfa);  
 **this**.deleteDeadState();  
 **this**.**keywords** = **new** HashSet<>();  
 System.***out***.println(**"DFA States: "**+**this**.getNodes().size());  
 System.***out***.println(**"Patterns (F): "**+**this**.getFinished().size());  
 System.***out***.println(**"Start: "**+**this**.getStart());  
 **this**.**io** = **new** LookAheadBufferedLexer();  
 **this**.**keywords** = **new** HashSet<>();  
 **this**.**operands** = **new** HashSet<>();  
 **this**.**aliases** = **new** HashMap<>();  
 **this**.**prevTok** = **null**;  
 }  
  
  
 **public** DFALexer(DFA dfa, **int** bsize){  
 **super**(dfa,**true**);*//set Minimization for lexer true.* **this**.deleteDeadState();  
 System.***out***.println(**"MinDFA States: "**+**this**.getNodes().size());  
 System.***out***.println(**"Patterns (F): "**+**this**.getFinished().size());  
 System.***out***.println(**"Start: "**+**this**.getStart());  
 **this**.**io** = **new** LookAheadBufferedLexer(bsize);  
 **this**.**keywords** = **new** HashSet<>();  
 **this**.**operands** = **new** HashSet<>();  
 **this**.**aliases** = **new** HashMap<>();  
 **this**.**prevTok** = **null**;  
 }  
  
 **public** DFALexer(DFA dfa){  
 **super**(dfa,**true**);*//set Minimization for lexer true.* **this**.deleteDeadState();  
 System.***out***.println(**"MinDFA States: "**+**this**.getNodes().size());  
 System.***out***.println(**"Patterns (F): "**+**this**.getFinished().size());  
 System.***out***.println(**"Start: "**+**this**.getStart());  
 **this**.**io** = **new** LookAheadBufferedLexer();  
 **this**.**keywords** = **new** HashSet<>();  
 **this**.**operands** = **new** HashSet<>();  
 **this**.**aliases** = **new** HashMap<>();  
 **this**.**prevTok** = **null**;  
 }  
  
 @Override  
 **public void** setKeywords(Set<String> kws){  
 **this**.**keywords** = kws;  
 }  
  
 @Override  
 **public** Set<String> getKeywords(){  
 **return keywords**;  
 }  
  
 @Override  
 **public void** setOperands(Set<String> operands) {  
 **this**.**operands** = operands;  
 }  
  
 @Override  
 **public** Set<String> getOperands() {  
 **return operands**;  
 }  
  
 @Override  
 **public void** setAliases(Map<String, String> aliases) {  
 **this**.**aliases** = aliases;  
 }  
  
 @Override  
 **public** Map<String, String> getAliases() {  
 **return aliases**;  
 }  
  
 @Override  
 **public** Token recognize(InputStream f) **throws** IOException {  
 **char** cur = (**char**)**io**.getch(f);  
 **while**(cur == **' '** || cur == **'\t'** || cur == **'\n'** || cur == **'\r'**) {  
 **if** (cur == **'\n'**) {  
 **io**.setCol(0);  
 }  
 cur = (**char**) **io**.getch(f);  
 }  
 **if**(((**int**)cur) == 65535)  
 **return new** Token(**"$"**,**"$"**,**'t'**);  
 Vertex s = **this**.**start**;  
 LinkedStack<Vertex> states = **new** LinkedStack<>();  
 states.push(s);  
 StringBuilder sb = **new** StringBuilder();  
 sb.append(cur);  
 **while**(((**int**)cur) != 65535 && !s.isDead()){*//while not EOF or not deadState.* s = moveTo(s,cur);  
 **if**(s == **null** || s.isDead()){  
 *//System.out.println("Lexeme at ("+io.getLine()+":"+io.getCol()+") :: "+sb.toString());* String err = sb.toString();  
 **while**(s == **null** || !s.isFinish()){  
 **if**(sb.length() == 0) {  
 **prevTok** = **new** Token(**"Unrecognized "**, **"Error at ("** + **io**.getLine() + **":"** + **io**.getCol() + **") :: Unrecognized token: "** + err + **"\n"**, **'e'**);  
 **return prevTok**;  
 }  
 cur = sb.charAt(sb.length() - 1);  
 sb.deleteCharAt(sb.length() - 1);  
 **io**.ungetch(cur);  
 s = states.top();  
 states.pop();  
 }  
 **if**(s.getValue().equals(**"id"**) && **keywords**.size() > 0 && **keywords**.contains(sb.toString())) {  
 **prevTok** = **new** Token(sb.toString(), sb.toString(), **'t'**);  
 **return prevTok**;  
 }  
 **else if**(**prevTok** == **null** || (**operands**.size() > 0 && **operands**.contains(**prevTok**.getName())) ){  
 **prevTok** = **new** Token(s.getValue(), sb.toString(), **'t'**);  
 **return prevTok**;  
 }  
 **else if**(**prevTok** != **null** && **operands**.size() > 0){  
 **prevTok** = **new** Token(**aliases**.get(s.getValue()), sb.toString(), **'t'**);  
 **return prevTok**;  
 }  
 **else** {  
 **prevTok** = **new** Token(s.getValue(), sb.toString(), **'t'**);  
 **return prevTok**;  
 }  
 }  
 **else** {  
 cur = (**char**)**io**.getch(f);  
 sb.append(cur);  
 states.push(s);  
 }  
 }  
 **if**(sb.length() > 0){  
 *//System.out.println("Lexeme at ("+io.getLine()+":"+io.getCol()+") :: "+sb.toString());* **if**((**int**)cur == 65535)  
 sb.deleteCharAt(sb.length() - 1);*//remove redundant read EOF ch.* **if**(s.isFinish()) {  
 **if**(s.getValue().equals(**"id"**) && **keywords**.size() > 0 && **keywords**.contains(sb.toString())) {  
 **prevTok** = **new** Token(sb.toString(), sb.toString(), **'t'**);  
 **return prevTok**;  
 }  
 **else if**(**prevTok** == **null** || (**operands**.size() > 0 && **operands**.contains(**prevTok**.getName())) ){  
 **prevTok** = **new** Token(s.getValue(), sb.toString(), **'t'**);  
 **return prevTok**;  
 }  
 **else if**(**prevTok** != **null** && **operands**.size() > 0){  
 **prevTok** = **new** Token(**aliases**.get(s.getValue()), sb.toString(), **'t'**);  
 **return prevTok**;  
 }  
 **else** {  
 **prevTok** = **new** Token(s.getValue(), sb.toString(), **'t'**);  
 **return prevTok**;  
 }  
 }  
 **else** {  
 **prevTok** = **new** Token(**"Unrecognized"**, **"Error at ("** + **io**.getLine() + **":"** + **io**.getCol() + **") :: Unrecognized token: "** + sb.toString() + **"\n"**, **'e'**);  
 **return prevTok**;  
 }  
 }  
 **else** {  
 **prevTok** = **new** Token(**"$"**, **"$"**, **'t'**);  
 **return prevTok**;  
 }  
 }  
  
 **private** Vertex moveTo(Vertex v, **char** c){  
 List<Pair<Vertex,Character>> k = **tranTable**.keySet().stream().filter(x -> x.getV1().equals(v) && x.getV2() == c).collect(Collectors.*toList*());  
 **return** k.size() > 0 ? **tranTable**.get(k.get(0)) : **null**;  
 }  
  
 **public** Token generateError(String s1, String s2){  
 **return new** Token(**"Unrecognized"**,**"Error at ("**+**io**.getLine()+**":"**+**io**.getCol()+**") :: Expected token: "**+s1+**" but actual: "**+s2+**"\n"**,**'e'**);  
 }  
  
 @Override  
 **public void** reset() {  
 **io**.setCol(0);  
 **io**.setLine(1);  
 **io**.clear();  
 **this**.**prevTok** = **null**;  
 }  
}

Листинг 4. Лексический анализатор, с обработкой ключевых слов и унарных операторов.

Теперь, перейдем к вычислению множеств L,R, Lt, Rt для нетерминальных символов грамматики. Множества L и R вычисляются с помощью методов экземпляра *getLMap* и *getRMap* класса Grammar соответственно. Они вычисляют множества L и R для каждого нетерминала, используя методы *getL\_i* и *getR\_i*, которым передаётся нетерминальный символ грамматики. Код вышеописанных методов дан далее.

*//Compute L(U) for U non-term***private** Set<GrammarSymbol> getL\_i(String header){  
 String h = header;  
 Set<GrammarSymbol> Ls = **new** HashSet<>();  
 LinkedStack<GrammarString> rules = **new** LinkedStack<>();  
 rules.addAll(**P**.get(h));  
 **while**(!rules.isEmpty()){  
 GrammarString rule = rules.top();  
 rules.pop();  
 GrammarSymbol si = rule.getSymbols().get(0);  
 **if**(!Ls.contains(si)){  
 Ls.add(si);  
 **if**(si.getType() == **'n'** && ! si.getVal().equals(h)){*//if it is a non-terminal and not equals to header.* rules.addAll(**P**.get(si.getVal()));  
 h = si.getVal();*//new rule with alternatives for new symbol si* }  
 }  
 }  
 **return** Ls;  
}  
  
*//Compute R(U) for U non-term***private** Set<GrammarSymbol> getR\_i(String header){  
 String h = header;  
 Set<GrammarSymbol> Rs = **new** HashSet<>();  
 LinkedStack<GrammarString> rules = **new** LinkedStack<>();  
 rules.addAll(**P**.get(h));  
 **while**(!rules.isEmpty()){  
 GrammarString rule = rules.top();  
 rules.pop();  
 GrammarSymbol si = rule.getSymbols().get(rule.getSymbols().size() - 1);  
 **if**(!Rs.contains(si)){  
 Rs.add(si);  
 **if**(si.getType() == **'n'** && ! si.getVal().equals(h)){*//if it is a non-terminal and not equals to header.* rules.addAll(**P**.get(si.getVal()));  
 h = si.getVal();*//new rule with alternatives for new symbol si* }  
 }  
 }  
 **return** Rs;  
}  
  
**public** Map<String,Set<GrammarSymbol>> getLMap(){  
 Map<String,Set<GrammarSymbol>> L = **new** HashMap<>();  
 Set<String> NT = **this**.**N**;  
 **for**(String h: NT){  
 L.put(h,getL\_i(h));  
 }  
 **return** L;  
}  
  
**public** Map<String,Set<GrammarSymbol>> getRMap(){  
 Map<String,Set<GrammarSymbol>> R = **new** HashMap<>();  
 Set<String> NT = **this**.**N**;  
 **for**(String h: NT){  
 R.put(h,getR\_i(h));  
 }  
 **return** R;  
}

Листинг 5. Методы экземпляра класса Grammar для вычисления множеств L(U), R(U) для каждого нетерминального символа U.

Вычислив множества L и R, на их основе вычисляются множества крайних левых терминальных символов (Lt) и крайних правых терминальных символов (Rt) для каждого нетерминального символа грамматики, с помощью соответствующих методов экземпляра класса Grammar: *getLeftTermMap* и *getRightTermMap*.

*//for each Non-term U compute Lt(U) based on L(U).***public** Map<String,Set<String>> getLeftTermMap(Map<String,Set<GrammarSymbol>> L){  
 Map<String,Set<String>> res = **new** HashMap<>();  
 Set<String> NT = **this**.**N**;  
 LinkedStack<String> S = **new** LinkedStack<>();  
 **for**(String N: NT){  
 S.push(N);  
 res.put(N,**new** ObservableHashSet<String>());  
 }  
 **while**(!S.isEmpty()){  
 String p = S.top();  
 S.pop();  
 Set<GrammarString> bodies = **this**.**P**.get(p);  
 **for**(GrammarString str : bodies){  
 **if**(str.getSymbols().size() == 1){  
 GrammarSymbol s = str.getSymbols().get(0);  
 **if**(s.getType() != **'t'**)  
 **continue**;  
  
 **if**(res.get(p).add(s.getVal())){  
 **for**(GrammarSymbol sym : L.get(p)){  
 **if**(sym.getType() == **'n'** && ! sym.getVal().equals(p))  
 ((ObservableHashSet<String>) res.get(sym.getVal())).attach((ObservableHashSet<String>) res.get(p));  
 }  
 }  
 }  
 **else**{  
 GrammarSymbol s = str.getSymbols().get(0);  
 **if**(s.getType() != **'t'**)  
 s = str.getSymbols().get(1);  
 **if**(s.getType() != **'t'**)  
 **continue**;  
 **if**(res.get(p).add(s.getVal())){  
 **for**(GrammarSymbol sym : L.get(p)){  
 **if**(sym.getType() == **'n'** && ! sym.getVal().equals(p))  
 ((ObservableHashSet<String>) res.get(sym.getVal())).attach((ObservableHashSet<String>) res.get(p));*//Lt(p) inherits from p in L(p)* }  
 }  
 }  
 }  
 }  
 **return** res;  
}  
  
*//for each Non-term U compute Rt(U) based on R(U).***public** Map<String,Set<String>> getRightTermMap(Map<String,Set<GrammarSymbol>> R){  
 Map<String,Set<String>> res = **new** HashMap<>();  
 Set<String> NT = **this**.**N**;  
 LinkedStack<String> S = **new** LinkedStack<>();  
 **for**(String N: NT){  
 S.push(N);  
 res.put(N,**new** ObservableHashSet<String>());  
 }  
 **while**(!S.isEmpty()){  
 String p = S.top();  
 S.pop();  
 Set<GrammarString> bodies = **this**.**P**.get(p);  
 **for**(GrammarString str : bodies){  
 **if**(str.getSymbols().size() == 1){  
 GrammarSymbol s = str.getSymbols().get(0);  
 **if**(s.getType() != **'t'**)  
 **continue**;  
  
 **if**(res.get(p).add(s.getVal())){  
 **for**(GrammarSymbol sym : R.get(p)){  
 **if**(sym.getType() == **'n'** && ! sym.getVal().equals(p))  
 ((ObservableHashSet<String>) res.get(sym.getVal())).attach((ObservableHashSet<String>) res.get(p));  
 }  
 }  
 }  
 **else**{  
 GrammarSymbol s = str.getSymbols().get(str.getSymbols().size() - 1);  
 **if**(s.getType() != **'t'**)  
 s = str.getSymbols().get(str.getSymbols().size() - 2);  
 **if**(s.getType() != **'t'**)  
 **continue**;  
 **if**(res.get(p).add(s.getVal())){  
 **for**(GrammarSymbol sym : R.get(p)){  
 **if**(sym.getType() == **'n'** && ! sym.getVal().equals(p))  
 ((ObservableHashSet<String>) res.get(sym.getVal())).attach((ObservableHashSet<String>) res.get(p));*//Lt(p) inherits from p in L(p)* }  
 }  
 }  
 }  
 }  
 **return** res;  
}

Листинг 6. Вычисление множеств Lt(U) и Rt(U) для каждого нетерминального символа U грамматики.

Метод экземпляра класса Grammar *getSpanningGrammar* возвращает новую остовную грамматику G\_s.

*//Works only for Operator Grammar***public** Grammar getSpanningGrammar(){  
 **if**(!isOperatorGrammar()){  
 System.***out***.println(**"Error. Grammar is not operator Grammar!"**);  
 **return null**;  
 }  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 Set<String> NN = **new** HashSet<>();  
 String start = getStart();  
 NN.add(start);  
 Set<String> p = **this**.**P**.keySet();  
 Set<GrammarString> nrules = **new** HashSet<>();  
 **for**(String r : p){  
 Set<GrammarString> rules = **P**.get(r);  
 **for**(GrammarString str: rules){  
 List<GrammarSymbol> symbols = **new** ArrayList<>();  
 **for**(GrammarSymbol sym : str.getSymbols()){  
 GrammarSymbol ns = **new** GrammarSymbol(sym.getType(),sym.getType() == **'t'** ? sym.getVal() : start);  
 symbols.add(ns);  
 }  
 **if**(symbols.size() > 1 || (symbols.get(0).getType() == **'t'**))  
 nrules.add(**new** GrammarString(symbols));  
 }  
 }  
 newP.put(start,nrules);  
 **return new** Grammar(**T**,NN,newP,**this**.**S**,**this**.**E**,**lex\_rules**,**this**.**keywords**,**this**.**operands**,**this**.**aliases**);

}

Листинг 7. Получение остовной грамматики G\_s из исходной операторной грамматики G.

Тип OperatorPresedenceRelations представляет собой матрицу отношений операторного предшествования для любых двух терминальных символов грамматики. Данный тип содержит матрицу в виде двумерного массива символов *matrix*. Так как каждая строка и каждый столбец данной матрицы помечен строковым именем терминала, то для преобразования строки к числу, т.е. к индексу массива, тип также хранит отображение из имён терминалов к индексам массива *matrix* в поле *mIdx*. Строка и столбец с одинаковым номером (индексом) имеют одинаковые метки. Далее приведён вышеописанного типа.

**package** ru.osipov.labs.lab4.parsers.generators;  
  
**import** java.util.Map;  
  
**public class** OperatorPresedenceRelations {  
 **private char**[][] **matrix**;  
 **private** Map<String,Integer> **idx**;  
  
 **public** OperatorPresedenceRelations(**char**[][] matrix,Map<String,Integer> mIdx){  
 **this**.**idx** = mIdx;  
 **this**.**matrix** = matrix;  
  
 }  
  
 **public char**[][] getMatrix() {  
 **return matrix**;  
 }  
  
 **public** Map<String, Integer> getMatrixIndices() {  
 **return idx**;  
 }  
}

Листинг 8. Класс OperatorPresedenceRelations.

Тип OPParserGenerator строит таблицу отношений операторного предшествования и возвращает её в виде типа OperatorPresedenceRelations. Также данный тип проводит индексацию правил грамматики, и сохраняет полученную информацию в типе ReduceIndexer. Информация необходим для операции свёртки. Ниже дан код классов OPParserGenerator и ReduceIndexer.

**package** ru.osipov.labs.lab4.parsers.generators;  
  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab2.grammars.GrammarString;  
**import** ru.osipov.labs.lab2.grammars.GrammarSymbol;  
  
**import** java.util.HashMap;  
**import** java.util.List;  
**import** java.util.Map;  
**import** java.util.Set;  
  
**public class** OPParserGenerator {  
  
 *//for reduce step.* **public** ReduceIndexer getIndicesOfRules(Grammar G){  
 Map<String,Integer> indices = **new** HashMap<>();  
 Map<Integer,String> parents = **new** HashMap<>();  
 Set<String> NT = G.getNonTerminals();  
 **int** i = 1;  
 **for**(String N : NT) {  
 **for** (GrammarString str : G.getProductions().get(N)) {  
 indices.put(str.toString().replaceAll(“\\s+”,””), i);  
 parents.put(i,N);  
 i++;  
 }  
 }  
 **return new** ReduceIndexer(indices,parents);  
 }  
  
 **public** OperatorPresedenceRelations operatorPresedence(Grammar G){  
 Set<String> T = G.getTerminals();  
 **int** sz = T.size() + 1;  
 **char** [][] matrix = **new char** [sz][sz];  
 Map<String,Integer> idx = **new** HashMap<>();  
 **int** i = 0;  
 **for**(String term : T){  
 idx.put(term,i);  
 i++;  
 }  
 idx.put(**"$"**,i);*//A[i,j] = term where i == j.* String marker = **"$"**;  
 Map<String,Set<String>> Lt = G.getLeftTermMap(G.getLMap());  
 Map<String,Set<String>> Rt = G.getRightTermMap(G.getRMap());  
  
 Set<String> t1 = Lt.get(G.getStart());*//Lt(S)* Set<String> t2 = Rt.get(G.getStart());*//Rt(S)* **int** j = 0;  
 **for**(String term : t1){  
 j = idx.get(term);  
 matrix[i][j] = **'<'**;  
 }  
 j = idx.get(**"$"**);  
 **for**(String term : t2){  
 i = idx.get(term);  
 matrix[i][j] = **'>'**;  
 }  
 T.add(**"$"**);   
 Set<String> P = G.getProductions().keySet();  
   
 **for** (String h : P) {  
 Set<GrammarString> rules = G.getProductions().get(h);*//get rules with header h.* **for** (GrammarString rule : rules) {  
 List<GrammarSymbol> l = rule.getSymbols();

**if**(l.size() == 1)

**continue**;

**for** (**int** li = 0; li < l.size() - 2; li++) {*//scanning body.* GrammarSymbol ai = l.get(li);  
 **if**(ai.getType() == **'t'**){  
 GrammarSymbol bj = l.get(li + 1);  
 **if**(bj.getType() == **'n'**){*//located rule like aU where a = term, U = non-term* Set<String> Li = Lt.get(bj.getVal());*//Lt(U)* **int** idx1 = idx.get(ai.getVal());  
 **for**(String c : Li){  
 **int** idx2 = idx.get(c);  
 matrix[idx1][idx2] = **'<'**;  
 }  
 bj = l.get(li + 2);  
 **if**(bj.getType() == **'t'**){*//located rule like aUb where a,b = terms, U = non-term.* idx1 = idx.get(ai.getVal());*//index a* **int** idx2 = idx.get(bj.getVal());*//index b* matrix[idx1][idx2] = **'='**;  
 ai = l.get(li + 1);*//return U* Set<String> Ri = Rt.get(ai.getVal());*//Rt(U)* **for**(String c : Ri){*//for rule like Ub* idx1 = idx.get(c);  
 matrix[idx1][idx2] = **'>'**;  
 }  
 }  
 }  
 **else if**(bj.getType() == **'t'**){*//located rule ab.* **int** idx1 = idx.get(ai.getVal());  
 **int** idx2 = idx.get(bj.getVal());  
 matrix[idx1][idx2] = **'='**;  
 }  
 }  
 *//looking for rules Uai where U = non-term, ai = term.* **else if**(ai.getType() == **'n'**){  
 GrammarSymbol bj = l.get(li + 1);  
 **if**(bj.getType() == **'t'**){*//located rule Ua where U = non-term, a = term.* Set<String> Ri = Rt.get(ai.getVal());*//Rt(U).* **int** idx2 = idx.get(bj.getVal());  
 **int** idx1 = 0;  
 **for**(String c : Ri){  
 idx1 = idx.get(c);  
 matrix[idx1][idx2] = **'>'**;  
 }  
 }  
 }  
 }  
 *//Check last two symbols of body.* GrammarSymbol a = l.get(l.size() - 2);  
 GrammarSymbol b = l.get(l.size() - 1);  
 **if**(a.getType() == **'t'** && b.getType() == **'t'**){  
 **int** idx1 = idx.get(a.getVal());  
 **int** idx2 = idx.get(b.getVal());  
 matrix[idx1][idx2] = **'='**;  
 }  
 **else if**(a.getType() == **'n'** && b.getType() == **'t'**){  
 Set<String> Ri = Rt.get(a.getVal());  
 **int** idx2 = idx.get(b.getVal());  
 **int** idx1 = 0;  
 **for**(String c : Ri){  
 idx1 = idx.get(c);  
 matrix[idx1][idx2] = **'>'**;  
 }  
 }  
 **else if**(a.getType() == **'t'** && b.getType() == **'n'**){  
 Set<String> Li = Lt.get(b.getVal());  
 **int** idx1 = idx.get(a.getVal());  
 **int** idx2 = 0;  
 **for**(String c : Li){  
 idx2 = idx.get(c);  
 matrix[idx1][idx2] = **'<'**;  
 }  
 }  
 }  
 }  
 **return new** OperatorPresedenceRelations(matrix,idx);  
 }  
}

**package** ru.osipov.labs.lab4.parsers.generators;  
  
**import** java.util.Map;  
  
**public class** ReduceIndexer {  
 **private** Map<String,Integer> **states**;  
 **private** Map<Integer,String> **headers**;  
  
 **public** ReduceIndexer(Map<String,Integer> states, Map<Integer,String> headers){  
 **this**.**states** = states;  
 **this**.**headers** = headers;  
 }  
  
 **public** Map<String, Integer> getStates() {  
 **return states**;  
 }  
  
 **public** Map<Integer, String> getHeaders() {  
 **return headers**;  
 }  
}

Листинг 9. Классы OPParserGenerator и ReduceIndexer.

И наконец, класс ShiftReduceParser реализует алгоритм “сдвиг-свёртка”, работая с матрицей отношений операторного предшествования и остовной грамматикой, полученной из исходной. Данный тип принимает в качестве параметров конструктора два объекта: исходную грамматику Grammar и лексический анализатор, реализующий интерфейс ILexer.

**package** ru.osipov.labs.lab4.parsers;  
  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab3.lexers.ILexer;  
**import** ru.osipov.labs.lab3.lexers.Token;  
**import** ru.osipov.labs.lab3.trees.LinkedNode;  
**import** ru.osipov.labs.lab3.trees.LinkedTree;  
**import** ru.osipov.labs.lab4.parsers.generators.OPParserGenerator;  
**import** ru.osipov.labs.lab4.parsers.generators.OperatorPresedenceRelations;  
**import** ru.osipov.labs.lab4.parsers.generators.ReduceIndexer;  
  
**import** java.io.\*;  
**import** java.util.Iterator;  
**import** java.util.List;  
  
*//Bottom-up parser with shift-reduce.  
//Works with Operator Grammars***public class** ShiftReduceParser {  
 **private** OperatorPresedenceRelations **table**;  
 **private** ReduceIndexer **rIdx**;  
 **private** ILexer **lexer**;  
  
 **public** ShiftReduceParser(Grammar G, ILexer lexer){  
 OPParserGenerator gen = **new** OPParserGenerator();  
 **this**.**lexer** = lexer;  
 lexer.setKeywords(G.getKeywords());  
 lexer.setOperands(G.getOperands());  
 lexer.setAliases(G.getAliases());  
 **this**.**table** = gen.operatorPresedence(G);  
 **this**.**rIdx** = gen.getIndicesOfRules(G.getSpanningGrammar());  
 System.***out***.println(**"Parser was built."**);  
 *//printTableRelations(G);* }  
  
 **private void** printTableRelations(Grammar G){  
 System.***out***.println(**"Matrix was built."**);  
 System.***out***.print(**"\t\t"**);  
 System.***out***.print(**"| "**);  
 **for**(String t: G.getTerminals()){  
 System.***out***.print(t+**" | "**);  
 }  
 **for**(String rt: G.getTerminals()){  
 System.***out***.print(**"| "**+rt);  
 **for**(String ct : G.getTerminals()){  
 **int** r = **table**.getMatrixIndices().get(rt);  
 **int** c = **table**.getMatrixIndices().get(ct);  
 System.***out***.print(**" | "**+**table**.getMatrix()[r][c]);  
 }  
 System.***out***.println(**" |"**);  
 }  
 }  
  
 *//Bottom-up parser:: shift-reduce.  
 //Operates with operators Grammar.* **public** LinkedTree<Token> parse(Grammar G, String fname){  
 FileInputStream f;  
 **boolean** isParsed = **false**;  
 **try** {  
 f = **new** FileInputStream(**new** File(fname).getAbsolutePath());  
 InputStreamReader ch = **new** InputStreamReader(f);  
  
 LinkedStack<LinkedNode<Token>> S = **new** LinkedStack<>();  
 LinkedNode<Token> EOF = **new** LinkedNode<>();  
 EOF.setValue(**new** Token(**"$"**,**"$"**,**'t'**));  
 Iterator<LinkedNode<Token>> S\_iter = S.iterator();  
  
 Token tok = **lexer**.recognize(f);*//get token from the input.* S.push(EOF);*//Initial:: ($,tok). //Finish:: ($S,$).* LinkedNode<Token> A = **null**;*//current symbol on Stack.* LinkedStack<Integer> rules = **new** LinkedStack<>();  
 **int** nidx = 1;  
 **while**(**true**){  
 S\_iter = S.iterator();*//restart iterator.  
 //Get first term on the top of the stack.* A = S.top();  
 Token X = A.getValue();  
 System.***out***.println(**"S = "**+S);  
 System.***out***.println(**"Input token: "**+tok.getName());  
 **while**(X.getType() != **'t'** && S\_iter.hasNext()){*//If non-terms on the top of the stack => ignore them.* A = S\_iter.next();  
 X = A.getValue();  
 }  
 System.***out***.println(**"On Stack: "**+X.getName());  
 **if**(X.getName().equals(**"$"**) && tok.getName().equals(**"$"**)){  
 isParsed = **true**;  
 **break**;  
 }  
 **int** idx1 = **table**.getMatrixIndices().get(X.getName());  
 **int** idx2 = **table**.getMatrixIndices().get(tok.getName());  
 **char** rel = **table**.getMatrix()[idx1][idx2];  
 **if**(rel == **'<'** || rel == **'='**){*//shift* LinkedNode<Token> ci = **new** LinkedNode<>();  
 ci.setValue(tok);  
 S.push(ci);  
 tok = **lexer**.recognize(f);  
 }  
 **else if**(rel == **'>'**){*//reduce  
 //System.out.println("reduce");* LinkedNode<Token> parent = **new** LinkedNode<>();  
 LinkedNode<Token> child\_a = S.top();  
 **while**(child\_a.getValue().getType() != **'t'**){  
 S.pop();  
 child\_a.setParent(parent);  
 child\_a.setIdx(nidx);  
 nidx++;  
 parent.getChildren().add(child\_a);  
 child\_a = S.top();  
 }  
 child\_a.setParent(parent);  
 child\_a.setIdx(nidx);  
 nidx++;  
 S.pop();  
 parent.getChildren().add(child\_a);  
 **while**(!S.isEmpty()){  
 LinkedNode<Token> child\_b = S.top();  
 S.pop();  
 **if**(child\_b.getValue().getType() == **'n'**){*//add non-terminal too* child\_b.setParent(parent);  
 child\_b.setIdx(nidx);  
 nidx++;  
 parent.getChildren().add(child\_b);  
 **continue**;  
 }  
 **else**{  
 **int** r = **table**.getMatrixIndices().get(child\_b.getValue().getName());  
 **int** c = **table**.getMatrixIndices().get(child\_a.getValue().getName());  
 **if**(**table**.getMatrix()[r][c] == **'='**){*//is base row.* child\_b.setParent(parent);  
 child\_b.setIdx(nidx);  
 nidx++;  
 parent.getChildren().add(child\_b);  
 child\_a = child\_b;  
 }  
 **else**{  
 S.push(child\_b);  
 **break**;  
 }  
 }  
 }  
 List<LinkedNode<Token>> children = parent.getChildren();  
 StringBuilder b = **new** StringBuilder();  
 **for**(**int** li = children.size() - 1; li >= 0; li--){  
 b.append(children.get(li).getValue().getName());  
 }  
 System.***out***.println(**"Reduced: "**+b.toString());  
 Integer r\_i = **rIdx**.getStates().get(b.toString());  
 rules.push(r\_i);  
 String header = G.getStart();  
 *//System.out.println(header);* parent.setValue(**new** Token(header,header,**'n'**));  
 S.push(parent);  
 System.***out***.println(S);  
 }  
 **else**{*//syntax error.* **break**;  
 }  
 }  
 **lexer**.reset();*//reset column and line counter.* **if**(isParsed)  
 **return new** LinkedTree<Token>(S.top());  
 **else** {  
 **if**(tok.getType() != **'e'**) {  
 System.***out***.println(**lexer**.generateError(S.top().getValue().getName(),tok.getLexem()));  
 }  
 **else** System.***out***.println(tok);  
 }  
 **return null**;  
 }  
 **catch** (FileNotFoundException e){  
 System.***out***.println(**"File not found. Specify file to read"**);  
 **return null**;  
 } **catch** (IOException e) {  
 System.***out***.println(**"File is not available now."**);  
 **return null**;  
 }  
 }  
}

Листинг 10. Класс ShiftReduceParser.

**4. Результаты работы программы и краткие выводы.**

Для входного файла грамматики из Листинга 1 и файла S\_G\_lab4\_mod.txt, содержимое которое имеет вид:

{  
 id = id + id && 1 \* id <> 25.2E-11 ^ i ;  
 id = id ^ 25 ;  
 {  
 id = (25.34 <= 25 + id \* id) || (a < b)  
 }  
}.

программа выдаёт следующее дерево остовного разбора грамматики.

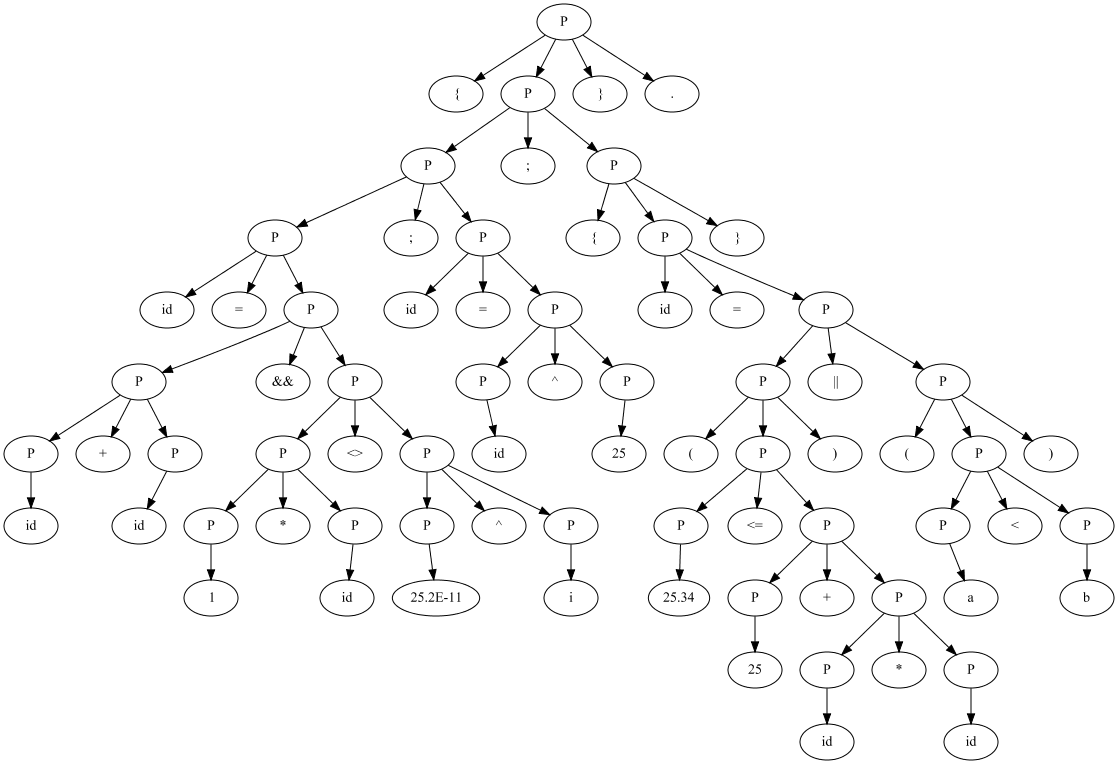


Рис. 1. Дерево разбора для входных данных.

Программа генерирует остовный вывод разбора грамматики Ниже дан код тестов, сравнивающее время выполнения разбора и количество вершин дерева разбора данным анализатором и анализатором из предыдущей лабораторной работы (LL-анализатор).

**package** ru.osipov.labs.lab4;  
  
**import** guru.nidi.graphviz.engine.Format;  
**import** guru.nidi.graphviz.engine.Graphviz;  
**import** org.mozilla.universalchardet.UniversalDetector;  
**import** org.junit.jupiter.api.Test;  
**import** org.junit.jupiter.api.extension.ExtendWith;  
**import** org.junit.runner.RunWith;  
**import** org.springframework.test.context.ContextConfiguration;  
**import** org.springframework.test.context.junit.jupiter.SpringExtension;  
**import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  
**import** ru.osipov.labs.lab1.structures.automats.CNFA;  
**import** ru.osipov.labs.lab1.structures.automats.DFA;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab2.jsonParser.SimpleJsonParser;  
**import** ru.osipov.labs.lab2.jsonParser.jsElements.JsonObject;  
**import** ru.osipov.labs.lab3.lexers.DFALexer;  
**import** ru.osipov.labs.lab3.lexers.Token;  
**import** ru.osipov.labs.lab3.lexers.generators.FALexerGenerator;  
**import** ru.osipov.labs.lab3.parsers.LLParser;  
**import** ru.osipov.labs.lab3.trees.LinkedTree;  
**import** ru.osipov.labs.lab4.parsers.ShiftReduceParser;  
  
**import** java.io.File;  
**import** java.io.FileInputStream;  
**import** java.io.IOException;  
**import** java.util.ArrayList;  
**import** java.util.HashMap;  
**import** java.util.List;  
**import** java.util.Map;  
  
@ExtendWith(SpringExtension.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(classes= TestOpParser.**class**)  
**public class** TestOpParser {  
 @Test  
 **public void** test1() **throws** IOException {  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 String dir = System.*getProperty*(**"user.dir"**) +**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 String s = System.*getProperty*(**"user.dir"**)+**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 s = s + **"input\\S\_G\_lab3\_mod.txt"**;  
 p = p+**"grammarJson\\G\_Lab4\_3.json"**;  
 SimpleJsonParser parser = **new** SimpleJsonParser();  
 JsonObject ob = parser.parse(p);  
 **assert** ob != **null**;  
 Grammar G = **new** Grammar(ob);  
 System.***out***.println(**"Source"**);  
 System.***out***.println(G);  
 Grammar GL = Grammar.*deleteLeftRecursion*(G);  
 GL = GL.deleteLeftFactor();  
 **assert** GL != **null**;  
  
  
 FALexerGenerator lg = **new** FALexerGenerator();  
 CNFA nfa = lg.buildNFA(G);  
 DFALexer lexer = **new** DFALexer(**new** DFA(nfa));  
 lexer.getImagefromStr(dir,**"lexer\_Test1"**);  
  
 LLParser sa1 = **new** LLParser(GL,lexer);  
 ShiftReduceParser sa2 = **new** ShiftReduceParser(G,lexer);  
  
 *//Test 1.* **long** current = System.*currentTimeMillis*();  
 LinkedTree<Token> t = sa1.parse(GL,s);  
 **assert** t != **null**;  
 **long** m1 = (System.*currentTimeMillis*()) - current;  
  
 current = System.*currentTimeMillis*();  
 LinkedTree<Token> t2 = sa2.parse(G,s);  
 **assert** t2 != **null**;  
 **long** m2 = (System.*currentTimeMillis*()) - current;  
 System.***out***.println(**"S\_G\_lab3\_mod.txt"**);  
 System.***out***.println(**"LL (Top-down): "**+m1);  
 System.***out***.println(**"Count of nodes: "**+t.getCount());  
 System.***out***.println(**"OP (Bottom-up): "**+m2);  
 System.***out***.println(**"Count of nodes: "**+t2.getCount());  
  
 Graphviz.*fromString*(t.toDot(**"LL\_parser\_test1"**)).render(Format.***PNG***).toFile(**new** File(dir+**"LL\_parser\_test1"**));  
 Graphviz.*fromString*(t2.toDot(**"OP\_parser\_test1"**)).render(Format.***PNG***).toFile(**new** File(dir+**"OP\_parser\_test1"**));  
 }  
  
 @Test  
 **public void** test2() **throws** IOException {  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 String dir = System.*getProperty*(**"user.dir"**) +**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 String s = System.*getProperty*(**"user.dir"**)+**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 s = s + **"input\\S\_G\_lab3\_mod2.txt"**;  
 p = p+**"grammarJson\\G\_Lab4\_3.json"**;  
 SimpleJsonParser parser = **new** SimpleJsonParser();  
 JsonObject ob = parser.parse(p);  
 **assert** ob != **null**;  
 Grammar G = **new** Grammar(ob);  
 System.***out***.println(**"Source"**);  
 System.***out***.println(G);  
 Grammar GL = Grammar.*deleteLeftRecursion*(G);  
 GL = GL.deleteLeftFactor();  
 **assert** GL != **null**;  
  
  
 FALexerGenerator lg = **new** FALexerGenerator();  
 CNFA nfa = lg.buildNFA(G);  
 DFALexer lexer = **new** DFALexer(**new** DFA(nfa));  
 *//lexer.getImagefromStr(dir,"lexer\_Test2");* LLParser sa1 = **new** LLParser(GL,lexer);  
 ShiftReduceParser sa2 = **new** ShiftReduceParser(G,lexer);  
  
  
 **long** current = System.*currentTimeMillis*();  
 LinkedTree<Token> t = sa1.parse(GL,s);  
 **assert** t != **null**;  
 **long** m1 = (System.*currentTimeMillis*()) - current;  
  
 current = System.*currentTimeMillis*();  
 LinkedTree<Token> t2 = sa2.parse(G,s);  
 **assert** t2 != **null**;  
 **long** m2 = (System.*currentTimeMillis*()) - current;  
 System.***out***.println(**"S\_G\_lab3\_mod2.txt"**);  
 System.***out***.println(**"LL (Top-down): "**+m1);  
 System.***out***.println(**"Count of nodes: "**+t.getCount());  
 System.***out***.println(**"OP (Bottom-up): "**+m2);  
 System.***out***.println(**"Count of nodes: "**+t2.getCount());  
  
 Graphviz.*fromString*(t.toDot(**"LL\_parser\_test2"**)).render(Format.***PNG***).toFile(**new** File(dir+**"LL\_parser\_test2"**));  
 Graphviz.*fromString*(t2.toDot(**"OP\_parser\_test2"**)).render(Format.***PNG***).toFile(**new** File(dir+**"OP\_parser\_test2"**));  
 }  
  
  
 @Test  
 **public void** test3() **throws** IOException {  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 String dir = System.*getProperty*(**"user.dir"**) +**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 String s = System.*getProperty*(**"user.dir"**)+**"\\src\\test\\java\\ru\\osipov\\labs\\lab4\\"**;  
 s = s + **"input\\G\_2\_27\_I.txt"**;  
 p = p+**"grammarJson\\G\_2\_27.json"**;  
 SimpleJsonParser parser = **new** SimpleJsonParser();  
 JsonObject ob = parser.parse(p);  
 **assert** ob != **null**;  
 Grammar G = **new** Grammar(ob);  
 System.***out***.println(**"Source"**);  
 System.***out***.println(G);  
 Grammar GL = Grammar.*deleteLeftRecursion*(G);  
 GL = GL.deleteLeftFactor();  
 **assert** GL != **null**;  
  
  
 FALexerGenerator lg = **new** FALexerGenerator();  
 CNFA nfa = lg.buildNFA(G);  
 DFALexer lexer = **new** DFALexer(**new** DFA(nfa));  
 lexer.getImagefromStr(dir,**"lexer\_Test3"**);  
  
 LLParser sa1 = **new** LLParser(GL,lexer);  
 ShiftReduceParser sa2 = **new** ShiftReduceParser(G,lexer);  
  
  
 **long** current = System.*currentTimeMillis*();  
 LinkedTree<Token> t = sa1.parse(GL,s);  
 **assert** t != **null**;  
 **long** m1 = (System.*currentTimeMillis*()) - current;  
  
 current = System.*currentTimeMillis*();  
 LinkedTree<Token> t2 = sa2.parse(G,s);  
 **assert** t2 != **null**;  
 **long** m2 = (System.*currentTimeMillis*()) - current;  
 System.***out***.println(**"G\_2\_27\_I.txt"**);  
 System.***out***.println(**"LL (Top-down): "**+m1);  
 System.***out***.println(**"Count of nodes: "**+t.getCount());  
 System.***out***.println(**"OP (Bottom-up): "**+m2);  
 System.***out***.println(**"Count of nodes: "**+t2.getCount());  
  
 Graphviz.*fromString*(t.toDot(**"LL\_parser\_test3"**)).render(Format.***PNG***).toFile(**new** File(dir+**"LL\_parser\_test3"**));  
 Graphviz.*fromString*(t2.toDot(**"OP\_parser\_test3"**)).render(Format.***PNG***).toFile(**new** File(dir+**"OP\_parser\_test3"**));  
 }  
}

Листинг 11. Код тестов.

Входом для первого теста является следующая цепочка:

{

id = id + id \* id <> 25.2E-11 ^ i ;

id = id ^ 25 ;

{

id = 25.34 <= 25 + id \* id

}

}.

Вход второго теста:

{

id = id + id \* id <> 25.2E-11 ^ i ;

id = id ^ 25 ;

{

id = 25.34 <= 25 + id \* id

} ;

a = 5 ;

b = a + a \* 2;

b = 25.12 ^ 3

}.

Вход третьего теста:

a = 12;

b = 33.125 ;

c = a + b \* 2 ;

d = a - ( b + 0.23 ) / 2 - 1

Программа выполнила данные тесты, и дала следующий результат.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер теста | Время LL | Время SR | Кол-ство вершин для LL | Кол-ство вершин для SR |
| 1 | 15 мс | 0 мс | 159 | 58 |
| 2 | 15 мс | 0 мс | 264 | 91 |
| 3 | 2 мс | 0 мс | 98 | 53 |

Табл. 1. Результаты тестов программы. LL – нисходящий нерекурсивный анализатор, SR – анализатор типа перенос-свертка (shift-reduce). Время дано в миллисекундах.

Как видно, получаемый анализатор для грамматики операторного предшествования строит остовный вывод, содержащий значительно меньше число вершин в дереве разбора, и работает быстрее, чем тот же анализатор для LL-грамматики.

**Список литературы и использованных источников.**

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции: В 2-х томах. Т.1.:Синтаксический анализ. –М.: Мир,1978.

2. Ахо А.В., Лам М.С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструменты. –М.: Вильямс, 2008.

3. Осипов О.К. Отчёт по лабораторной работе №3.

4. Осипов О.К. Отчёт по лабораторной работе №2.

5. Осипов О.К. Отчёт по лабораторной работе №1.

6. Молчанов А. Ю. Системное программное обеспечение. Учебник для вузов. 3-е изд. –Спб.: Питер, 2010. -400с.: ил.

7. Молчанов А. Ю. Системное программное обеспечение. Лабораторный практикум. – Спб.: Питер, 2005. - 284с.: ил.