Государственное образовательное учреждение высшего профессионального образования

|  |  |
| --- | --- |
| Gerb-BMSTU_01 | ***«Московский государственный технический университет  имени Н.Э. Баумана»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА

И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

**РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовому проекту на тему:**

**Самоприменимый генератор компиляторов методом рекурсивного спуска**

Студент \_\_\_\_\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** Копьев Е. В.

(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта \_\_\_\_\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** Коновалов А. В.

(Подпись, дата) (И.О.Фамилия)

Москва, 2015

**Содержание**

[**Введение** 3](#_Toc438461319)

[**1. Ключевые понятия и терминология** 5](#_Toc438461320)

[**2. Реализация первичной грамматики** 7](#_Toc438461321)

[**3. Лексический и синтаксический анализ первичной грамматики** 10](#_Toc438461322)

[**4. Метафункция PARSE** 12](#_Toc438461323)

[**5. Фаза синтеза** 14](#_Toc438461324)

[**6. Тестирование** 16](#_Toc438461325)

[**7. Инструкция для пользователей** 18](#_Toc438461326)

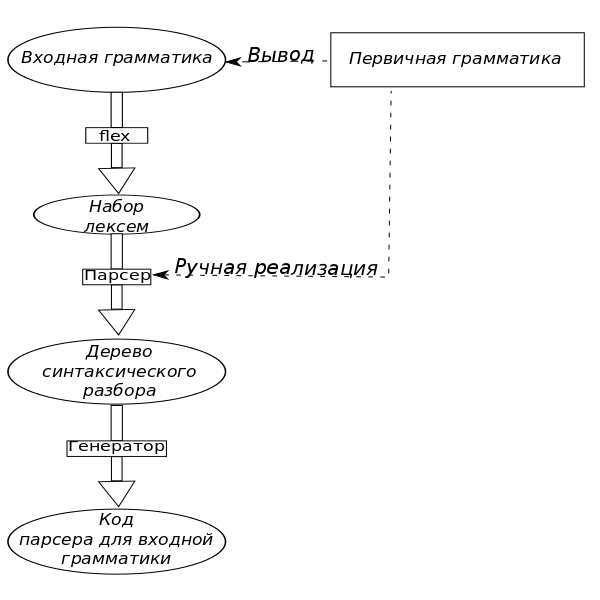
[**Заключение** 22](#_Toc438461327)

[**Литература** 23](#_Toc438461328)

**Введение**

Целью данного курсового проекта является реализация самоприменимого генератора компиляторов методом рекурсивного спуска. Как видно на рисунке 1, эту задачу можно разбить на несколько частей:

1. Выбор некоторого стандарта записи LL1 грамматик c семантическими правилами (обозначим множество таких предложений символом Ω).
2. Реализация первичной грамматики, в которой можно вывести все предложения из языка Ω.
3. Реализация лексера с использованием генератора лексических анализаторов flex.
4. Ручная реализация парсера по первичной грамматике методом рекурсивного спуска.
5. Разработка и реализация контейнерных классов для хранения дерева синтаксического разбора и промежуточного представления грамматики.
6. Реализация генератора синтаксических анализаторов.

Рисунок 1. Этапы реализации.

После реализации данной программы, появится возможность генерировать синтаксические анализаторы с поддержкой семантических правил для ряда грамматик. Предполагается также провести несколько этапов раскрутки. Каждый из этапов раскрутки представляет собой следующую последовательность действий:

1. Используя грамматику предыдущего этапа раскрутки (для которой был сгенерирован парсер), выводится новая грамматика с расширенными возможностями.
2. Используя программу, генерируется парсер для расширенной версии грамматики.

**1. Ключевые понятия и терминология**

Ввиду того что в дальнейшем будет использоваться большое количество терминов, необходимо заранее описать их значения.

Рассматриваются следующие фазы компиляции:

1. Лексический анализ — процесс разбиения входного текста на лексемы.
2. Синтаксический анализ — процесс верификации структуры входного предложения, на основе грамматики языка, и построения промежуточного представления.
3. Семантический анализ — процесс верификации промежуточного представления, на основе заранее заданных правил.
4. Оптимизация — процесс оптимизации промежуточного представления.
5. Синтез — процесс преобразования промежуточного представления в код, написанный на целевом языке.

В рамках данного проекта будут рассмотрены все фазы кроме оптимизации. Парсером называют программу, выполняющую синтаксический анализ. Лексером — программу, выполняющую лексический анализ.

Контекстно-свободной грамматикой будем называть кортеж (T, N, S, R) в котором T — множество терминальных символов, N — множество нетерминальных символов, S — аксиома или начальное правило, R — множество правил. При этом в левой части правила стоит один нетерминальный символ, а в правой части правила последовательность из терминальных, нетерминальных символов, а также специальных символов операций. В расширенной форме Бэкуса-Наура рассматриваются следующие операции: конкатенация, альтернатива, опциональное выражение, итерация. В дальнейшем, для записи грамматик, будем использовать нотацию Вирта. В ней операция конкатенации обозначается последовательной записью конкатенируемых конструкций, альтернатива – вертикальной чертой, опциональное выражение заключается в квадратные скобки, а выражение итерации в фигурные. Множителем будем называть идентификатор (то есть терминальный или нетерминальный символ), выражение в круглых скобках, итерацию или опциональное выражение. Выражением будем называть последовательность слагаемых, разделенных символами ‘ | ‘. Слагаемым будем называть последовательность множителей. Первичной будем называть ту грамматику, для которой парсер был написан вручную. Грамматикой реализации называется та грамматика, для которой был реализован парсер. Таким образом, первичная грамматика является грамматикой реализации. Входной грамматикой называется грамматика, подающаяся на вход генератору компиляторов. В нашем случае первичная и входная грамматика – это LL1 грамматики, записанные в нотации Вирта. С помощью грамматики можно задать язык, то есть множество последовательностей, состоящих из терминальных символов. Целевым языком будем называть язык, в который преобразуются промежуточное представление на этапе синтеза.

Семантическое правило — операция, выполняемая над синтезируемыми атрибутами символов грамматики. Синтезируемый атрибут символа грамматики — значение некоторого фиксированного типа. Значения синтезируемых атрибутов терминальных символов определяется в процессе лексического анализа. Значения синтезируемых атрибутов нетерминальных символов вычисляются в процессе синтаксического анализа.

**2. Реализация первичной грамматики**

Первым шагом в реализации программы является описание первичной грамматики. Для этого необходимо задать множество Ω. Пусть Ω - множество LL1 грамматик, записанных в нотации Вирта, с поддержкой семантических правил. Далее на основе языка Ω можно описать первичную LL1 грамматику в нотации Вирта, которая будет описывать язык, состоящий из таких же грамматик (см. листинг 1). Таким образом, запись первичной грамматики также является предложением в языке Ω, описанном первичной грамматикой.

*Листинг 1. Первичная грамматика.*

%term <\_\_string\_\_> @SEMRULE, <\_\_string\_\_> @TYPE,

<\_\_string\_\_> @NONTERM, <\_\_string\_\_> @TERM.

grammar = [@SEMRULE] init rule {rule} [@SEMRULE].

init = [term\_decl] [nonterm\_decl].

term\_decl = '%term' [attr] @TERM {',' [attr] @TERM} '.' .

nonterm\_decl = '%nonterm' [attr] @NONTERM

{ ',' [attr] @NONTERM } '.' .

attr = '<' @TYPE '>' .

rule = @NONTERM [@SEMRULE]'=' multi\_addendum '.' .

multi\_addendum = addendum {'|' addendum}.

addendum = factor [@SEMRULE] {factor [@SEMRULE] }.

factor = '[' multi\_addendum ']' |

'{' multi\_addendum '}' |

'(' multi\_addendum ')' |

ident | .

ident = @TERM | @NONTERM .

Далее подробно описывается структура этой грамматики и значения используемых символов. Для хранения семантических атрибутов символов будет использоваться объединение TYPES. Все лексемы будут иметь поле attr типа TYPES\* изначально равное NULL. В процессе лексического анализа для некоторых лексем в это поле будет записано соответствующее ее типу значение. Лексемы с тэгом TERM — это терминальные символы входной грамматики. Ключевое слово %term описывает семантические атрибуты символов, в которые будут преобразованы лексемы с тэгом TERM. Это преобразование задается с помощью поля mark, заданного в лексемах с тэгом TERM, и используется для того, чтобы сгенерированный парсер знал какие лексемы ему ожидать:

1. Вместо @SEMRULE будет ожидаться лексема SP\_SEMRULE (т. е. код между строками “%{” и “%}”).
2. Вместо @TERM будет ожидаться специальная лексема TERM, которую пользователь программы должен будет переопределить так, как нужно ему в зависимости от типа входной грамматики.
3. Вместо @NONTERM будет ожидаться лексема SP\_NONTERM, описывающая нетерминальные символы (т. е. выражения, начинающиеся с буквы и состоящие из последовательности маленьких латинских букв, цифр и символов нижнего подчеркивания).
4. Вместо @TYPE будет ожидаться лексема SP\_TYPE, описывающая поля объединения TYPES.
5. Вместо символа в одинарных кавычках будет ожидаться тот же символ без кавычек.

Ключевое слово %nonterm описывает типы семантических атрибуты нетерминальных символов входной грамматики (то есть возвращаемые значения функций сгенерированных методом рекурсивного спуска).

Упомянутые выше семантические правила представляют собой код на языке С++, а также предоставляют возможность ссылаться на синтезируемые атрибуты терминальных и нетерминальных символов. В процессе вывода может появиться необходимость передавать синтезируемый атрибут нетерминального символа, стоящего в левой части правила, родительским узлам. Для этого будет использоваться переменная $$. В процессе вывода эта переменная модифицируется в соответственных блоках семантических правил, а после достижению конца правила вывода эта переменная передается в родительский узел. Переданное в родительский узел значение будем называть семантическим атрибутом нетерминального символа. Для нумерации синтезируемых атрибутов терминальных и нетерминальных символов в грамматике, записанной в нотации Вирта, используются следующие правила:

1. Идентификатор начинается с символа ‘ $ ‘.
2. После этого указывается ноль или несколько символов ‘ ^ ‘. Каждый такой символ означает понижение уровня вложенности относительно текущего.
3. Далее указывается номер множителя на выбранном уровне вложенности.
4. Нумерация начинается с 1.
5. Нумеруются только множители, имеющие семантические атрибуты.

Рассмотрим пример, представленный в листинге 2. Пусть все символы в этом примере имеют семантические атрибуты. Тогда из семантического правила, расположенного после нетерминального символа G, можно сослаться к атрибуту символа G, используя конструкцию $1, а к атрибуту символа B, используя конструкцию $^1. Отметим также, что из данного семантического правила нельзя сослаться к атрибутам символа C, D или F.

*Листинг 2. Пример нумерации семантических атрибутов.*

A = B ( C | D ) { F | G %{...%} }.

Стоит отметить, что в процессе синтаксического анализа входной грамматики парсер всегда будет строить дерево синтаксического вывода, хранящее грамматику или некоторое выражение. Впоследствии, это дерево будет использоваться для генерации промежуточного представления входной грамматики, а уже по этому представлению будет генерироваться код парсера, в который будут копироваться семантические правила.

Далее представлен пример интерпретатора арифметических выражений, записанный на языке, описанном первичной грамматикой, который в дальнейшем будет использоваться для тестирования (см. листинг 3).

*Листинг 3. Пример входной грамматики с семантическими правилами.*

%{ %}

%term <\_\_int\_\_> @NUMBER.

%nonterm <\_\_int\_\_> expr, <\_\_int\_\_> addendum,

<\_\_int\_\_> factor.

program = statement @SEMICOLON program | .

statement = @PRINT out outlist.

outlist= @COMMA out outlist | .

out = expr %{cout<<$1<<endl;%}.

expr = addendum %{$$=$1;%} { @PLUS addendum %{$$+=$1;%}

| @MINUS addendum %{$$-=$1;%} }.

addendum = factor %{$$=$1;%} { @MULTI factor %{$$\*=$1;%}

| @DIVIDE factor %{$$/=$1;%} }.

factor %{ %} = @LPAREN expr @RPAREN %{$$=$1;%}

| @NUMBER %{$$=$1;%}

| @MINUS factor %{$$=-$1;%}

| @DIVIDE factor @COMMA factor

@COMMA factor @COMMA factor @DIVIDE

%{

if ($1 < 0)

$$ = $2;

else if ($1 == 0)

$$ = $3;

else

$$ = $4;

%}

| @MULTI factor %{$$=0; int N=0;%}

{@COMMA factor %{N++; if ($^1==N) $$=$1;%}} @MULTI.

%{ %}

**3. Лексический и синтаксический анализ первичной грамматики**

В рамках данного курсового проекта часто приходится генерировать лексические анализаторы, поэтому удобно воспользоваться генератором лексических анализаторов. Далее для генерации лексических анализаторов на языке С будет использоваться flex.

На этапе лексического анализа:

1. Верифицируются входные данные, на предмет некорректных лексем.
2. Задаются значения синтезируемых атрибутов терминальных символов.
3. Генерируется массив лексем.

В данном курсовом проекте лексемы классифицируются следующим образом:

1. Специальные служебные символы, необходимые для записи грамматики (нетерминальные символы входной грамматики, ключевые слова, типы, семантические правила, а также символы, используемые в нотации Вирта).
2. Терминальные символы входной грамматики (для первичной грамматики это специальные символы в кавычках и символы, начинающиеся с символа @).

Полученный массив лексем подается на вход синтаксического анализатора. На данном этапе:

1. Верифицируется синтаксическая структура.
2. Генерируется дерево синтаксического вывода.

В случае если текущая входная грамматика описывает LL1 грамматику в нотации Вирта и удовлетворяет ряду требований, то по дереву синтаксического разбора входной грамматики можно будет сгенерировать промежуточное представление. В данном проекте промежуточное представление будет описывать класс Grammar, который содержит в себе вектор правил грамматики, стартовое и финальное семантические правила, а также информацию о типах атрибутов терминальных и нетерминальных символов. Каждое правило грамматики, описанное классом Rule, представляет собой левую и правую часть. Нетерминальный символ в левой части правила описывает структура YYSTYPE с тэгом SP\_NONTERM. Выражение в правой части правила описывается классом MultiAddendum. Слагаемые и множители описываются классами Addendum и Factor соответственно. Каждый Factor содержит информацию в зависимости от типа множителя. Итерации, опциональные выражения или выражения в скобках содержат указатель на MultiAddendum; идентификатор содержит указатель на YYSTYPE. Вне зависимости от типа, Factor может содержать указатель на семантическое правило. Все эти классы строятся на основе дерева синтаксического вывода по специальным символам грамматики. После построения они используются для вычисления множеств FIRST входной грамматики. Множества FIRST сохраняются в соответствующих классах и становятся частью промежуточного представления. Используя промежуточное представление можно сгенерировать парсер. Далее будет более формально описан этот процесс.

**4. Метафункция PARSE**

Пусть E — пустая последовательность, a — терминальный символ, X — нетерминальный символ, Fi — множитель грамматики, Ai – слагаемое грамматики, M – последовательность слагаемых, sym — текущий токен, next\_token – функция, позволяющая получить следующий токен, FIRST — множество токенов с которых может начинаться регулярное выражение, SEMRULE — функция, сопоставляющая множителю семантическое правило. Тогда описанная в листингах (4-10) [1] метафункция PARSE позволяет сгенерировать парсер на основе метода рекурсивного спуска:

*Листинг 4. Преобразование пустого правила.*

PARSE( E ) : /\*do nothing\*/

*Листинг 5. Преобразование терминального символа.*

PARSE( a ) : {

if ( sym == a )

sym = next\_token( ) ;

else

report\_error( ) ;

}

}

*Листинг 6. Преобразование нетерминального символа.*

PARSE( X ) : {

X();

}

*Листинг 7. Преобразование конкатенации символов.*

PARSE( F1 … Fn ) : {

PARSE( F1 );

SEMRULE( F1 );

…

PARSE( Fn ) ;

SEMRULE( Fn );

}

*Листинг 8. Преобразование альтернативы.*

PARSE( A1 | … | An ) : {

switch( sym.tag )

case A1.tag:

PARSE( A1 );

…

case An.tag:

PARSE( An );

}

*Листинг 9. Преобразование итерации.*

PARSE( {M} ) : {

while ( sym \in FIRST( M ) )

PARSE( M );

}

*Листинг 10. Преобразование опциональных символов.*

PARSE( [M] ) : {

if ( sym \in FIRST( M ) )

PARSE( M );

}

**5. Фаза синтеза**

На основе промежуточного представления, используя метафункцию PARSE, генерируется класс парсер. В нем есть три публичных метода: конструктор, деструктор и функция parse, запускающая синтаксический анализ и возвращающая указатель на построенное дерево синтаксического разбора, т. е. указатель на класс Node. Стоит отметить тот факт, что в процессе синтаксического разбора выделяется динамическая память для хранения узлов дерева вывода, поэтому пользователь должен будет самостоятельно освободить выделенную память. За генерацию кода отвечает класс Generator. В конструкторе этого класса генерируется промежуточное представление входной грамматики, а также производится логирование данных, полученных на каждом этапе. За генерацию целевого кода отвечает метод generateCppParser. В нем генерируются три публичных метода описанных выше, по одному приватному методу на каждое правило входной грамматики, а также методы expectedToken и nextToken. Метод expectedToken проверяет тэг текущего токена, расположенного в переменной sym, на совпадение с параметром метода. В случае успеха вызывается метод nextToken, который заносит в переменную sym следующий токен, а в случае неудачи вызывается исключение.

Если вернуться к листингу 3, то можно заметить, что семантические правила можно помещать в различные части грамматики:

1. Если поместить правило в начало грамматики, то оно будет скопировано в самое начало функции parse, с которой начинается парсинг любого выражения. Такое правило может быть только одно.
2. Если поместить правило в конец грамматики, то оно будет скопировано в самый конец функции parse (то есть перед return). Такое правило также может быть только одно.
3. Если поместить правило после нетерминального символа в левой части правила, то оно будет скопировано в самое начало функции соответствующей данному правилу.
4. Также семантическое правило можно объявить в любом месте в правой части правила. В этом случае позиция семантического правила будет определяться ее позицией в правиле грамматики (см. листинг 7).

Стоит отметить, что на практике можно обойтись лишь семантическими правилами из пункта (4), но для большей наглядности были реализованы способы, описанные в пунктах (1-3). Перед копированием в целевой код семантические правила входной грамматики слегка модифицируются. Вместо идентификаторов атрибутов подставляются имена переменных. Для того чтобы обрабатывать идентификаторы атрибутов был реализован класс ManagerOfVariables, хранящий информацию об уровнях вложенности блоков языка С++, а они имеют ту же структуру, что и уровни вложенности в правилах входной грамматики. Из семантического правила удаляются все переводы строк. Далее полученная строка вставляется в код парсера, при этом перед ней вставляется отступ, а после нее символ перевода строки. Таким образом, семантические правила записываются в одну строчку и не форматируются. Перед строкой семантического правила и после нее вставляется ключевое слово #line, которое используется для перенаправления ошибок из сгенерированного файла в файл с входной грамматикой. Это упрощает отладку программы.

**6. Тестирование**

На основе грамматике из листинга 3 был сгенерирован парсер (cм. листинги 11,12). В результате синтаксического анализа конструкций из листинга 13, с использованием полученного класса, в консоль были выведены следующие значения: 1, 2, 3, 2, 7, -6, 5, 0. Также было получено дерево синтаксического разбора, соответствующее входной конструкции.

*Листинг 11. Фрагмент сгенерированного кода из метода parse\_factor, выполняющего синтаксический анализ оператора выбора /f1,f2,f3,f4/.*

expected\_token(parent->addChild(new Node(parent,sym)),

DIVIDE);

int value1 = parse\_factor(parent->addChild(new Node(parent,

"factor")));

expected\_token(parent->addChild(new Node(parent,sym)),

SP\_COMMA);

int value2 = parse\_factor(parent->addChild(new

Node(parent,"factor")));

expected\_token(parent->addChild(new Node(parent,sym)),

SP\_COMMA);

int value3 = parse\_factor(parent->addChild(new

Node(parent,"factor")));

expected\_token(parent->addChild(new Node(parent,sym)),

SP\_COMMA);

int value4 = parse\_factor(parent->addChild(new

Node(parent,"factor")));

expected\_token(parent->addChild(new Node(parent,sym)),

DIVIDE);

#line 18 "resources/inputGrammar"

if (value1 < 0)

result = value2;

else if (value1 == 0)

result = value3;

else result = value4;

#line 191 "results/GeneratedParser.cpp"

break;

*Листинг 12. Фрагмент сгенерированного кода из метода parse\_factor, выполняющего синтаксический анализ оператора выбора \*f1,f2,…,fn\*.*

expected\_token(parent->addChild(new Node(parent,sym)),

SP\_MULTI);

int value1 = parse\_factor(parent->addChild(new

Node(parent,"factor")));

#line 19 "resources/inputGrammar"

result=0; int N=0;

#line 200 "results/GeneratedParser.cpp"

while ( sym->tag==SP\_COMMA) {

expected\_token(parent->addChild(new Node(parent,sym)),

SP\_COMMA);

int value2 = parse\_factor(parent->addChild(new

Node(parent,"factor")));

#line 20 "resources/inputGrammar"

N++; if (value1==N) result=value2;

#line 206 "results/GeneratedParser.cpp"

}

expected\_token(parent->addChild(new Node(parent,sym)),

SP\_MULTI);

break;

*Листинг 13. Преобразование опциональных символов.*

PRINT /-1, 1, 2, 3/;

PRINT /0, 1, 2, 3/;

PRINT /1, 1, 2, (3/1)/;

PRINT 1 + 1, 3\*2 + 1;

PRINT -(7-2/2);

PRINT \*5, 1, 2, 3, 4, 5, 6\*;

PRINT \*9, 1, 2, 3, 4, 5, 6\*;

**7. Инструкция для пользователей**

Первым делом для того чтобы сгенерировать парсер необходимо записать входную LL1 грамматику в нотации Вирта, используя следующие соглашения:

1. Нетерминальные символы - это последовательность маленьких латинских букв, цифр и символов нижнего подчеркивания, которая начинается с буквы.
2. Левая и правая части правил разделяются символом ‘ = ‘.
3. Правило оканчивается символом ‘ . ‘.
4. Альтернатива обозначается символом ‘ | ‘.
5. Конкатенация обозначается последовательной записью конкатенируемых конструкций.
6. Опциональный символ заключается в квадратные скобки: ‘ [ ‘ и ‘ ] ‘.
7. Итерация обозначается фигурными скобками: ‘ { ‘ и ‘ } ‘.
8. Конкатенация обладает более высоким приоритетом, нежели альтернатива.
9. Расставить приоритеты операций можно с помощью круглых скобок.
10. Семантические правила заключаются в “ %{ “ и “ %} “.
11. Код семантических правил записывается на языке С++.
12. В семантических правилах нельзя использовать переменные, начинающиеся на result или value.
13. Семантические правила могут располагаться в начале грамматики, конце грамматики, после нетерминальных символов в левой части правила, в любом месте в правой части правила.
14. Аксиомой грамматики является первое записанное правило.

Далее необходимо сгенерировать лексер так, чтобы он был совместим с генератором. В файле lexer.l необходимо корректно описать значение лексем с тэгом TERM (см. листинг 14). Лексемы с тэгом TERM — это терминальные символы входного языка. Для различных регулярных выражений, которые будут интерпретироваться, как TERM необходимо задать значение поля mark. Значение поля mark — это тэг лексемы, в которую отобразится рассматриваемое подмножество лексем с тэгом TERM в сгенерированном парсере (то есть те значения, которые будет ожидать парсер в функции expectedToken).

*Листинг 14. Пример описания терминальных символов входной грамматики (фрагмент кода из файла lexer.l).*

void createStrAttr(YYSTYPE \*val, char\* text) {

val->attr = new TYPES;

val->attr->\_\_string\_\_ = new string(text);

}

…

@SEMRULE {

yylval->tag = TERM;

yylval->mark = SP\_SEM\_RULE;

createStrAttr(yylval, yytext);

return TERM;

}

…

'\[' {

yylval->tag = TERM;

yylval->mark=SP\_L\_SQ\_BRACKET;

createStrAttr(yylval, yytext);

return TERM;

}

Для всех используемых в поле mark тэгов необходимо описать лексемы. Для этого тэг лексемы должен быть внесен в перечисление yytokentype (см. листинг 15), а ее строковое представление в массив строк Domains на позицию равную ее тэгу (см. листинг 16).

*Листинг 15. Перечисление тэгов yytokentype (фрагмент кода из файла lexer.h).*

enum yytokentype {

…

SP\_SEM\_RULE = 3,

…

SP\_L\_SQ\_BRACKET = 6,

…

};

*Листинг 16. Массив имен лексем (фрагмент кода из файла lexer.l).*

const char\* Domains[] = {

…

"SP\_SEM\_RULE",

/\*4\*/

…

/\*6\*/

"SP\_L\_SQ\_BRACKET",

…

};

Для задания семантических атрибутов символов входной грамматики используются ключевые слова %term и %nonterm. После %term указывается терминальный символ грамматики реализации с тэгом TERM, хотя вместо него подразумевается символ входной грамматики, в который он будет отображен. Далее записывается семантический атрибут в угловых скобках. Семантический атрибут представляет собой поле объединения TYPES. Если таких символов больше одного, то они перечисляются через запятую, после чего ставится точка (см. листинг 3). Аналогично используется ключевое слово %nonterm, только вместо символов с тэгом TERM рассматриваются символы с тэгом SP\_NONTERM, которые являются нетерминальными символами входной грамматики.

Все типы семантических атрибутов, используемые при объявлении, должны быть указаны в объединении TYPES (см. листинг 17). Также должна быть реализована соответствующая объединению TYPES лексема SP\_TYPE. В поле attr\_type этой лексемы нужно записать строковое представление рассматриваемого типа целевого языка (см. листинг 18).

*Листинг 17. Объединение TYPES (фрагмент кода из файла lexer.h).*

union TYPES {

string \*\_\_string\_\_;

int \*\_\_int\_\_;

char \*\_\_char\_\_;

};

*Листинг 18. Лексема SP\_TYPE (фрагмент кода из файла lexer.l).*

\_\_int\_\_ {

yylval->tag = SP\_TYPE;

createStrAttr(yylval, yytext);

yylval->attr\_type = new string("int");

return SP\_TYPE;

}

\_\_char\_\_ {

yylval->tag = SP\_TYPE;

createStrAttr(yylval, yytext);

yylval->attr\_type = new string("char");

return SP\_TYPE;

}

\_\_string\_\_ {

yylval->tag = SP\_TYPE;

createStrAttr(yylval, yytext);

yylval->attr\_type = new string("string");

return SP\_TYPE;

}

Далее необходимо скомпилировать исходный код лексера и генератора, а после запустить полученный исполняемый файл. Введите флаг генерации кода *g*, путь к файлу с входной грамматикой, имя класса, путь к генерируемому файлу. В ответ на запрос из листинга 19 программа сгенерирует файлы *Parser.h* и *Parser.cpp*в папке *results*, содержащие класс GeneratedParser, используя входную грамматику из указанного файла. Стоит отметить, что полученные файлы зависят от файлов Node.h и Node.cpp, а также ожидают заголовочный файл с объявлениями лексем по пути lexer/lexer.h.

*Листинг 19. Ожидаемый текстовый ввод.*

Enter mode-flag:

**g**

Enter path to file with grammar:

**resources/firstGrammar**

Enter name of parse-class:

**GeneratedParser**

Enter path to file:

**results/Parser**

The program is completed successfully!

**Заключение**

В рамках данной курсовой работы был реализован самоприменимый генератор компиляторов методом рекурсивного спуска. Он позволяет генерировать парсеры на языке С++ для LL1 грамматик, записанных в нотации Вирта. Более того, была реализована поддержка семантических правил на языке С++. Реализованный генератор компиляторов является самоприменимым в том смысле, что с его помощью можно сгенерировать парсер для первичной грамматики идентичный реализованному вручную. Это позволяет выполнять раскрутку данного генератора компиляторов.

**Литература**

1. Лекционный материал Скоробогатова С. Ю. по курсу конструирования компиляторов, 2015.