Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Московский государственный технический университет имени Н.Э. Баумана
(МГТУ им. Н.Э. Баумана)

Факультет: «Информатика и системы управления»
Кафедра: «Теоретическая информатика и компьютерные технологии»

Расчетно-пояснительная записка
к курсовому проекту

Тестер синтаксических анализаторов

Руководитель курсового проекта: _____ (А.В. Коновалов)
(подпись, дата)

Исполнитель курсового проекта,
студент группы ИУ9-72: _____ (М.С. Макаров)
(подпись, дата)

Москва, 2016

Содержание

Введение

В настоящее время многократно возросло использования всевозможных языков высокого уровня. Для того, чтобы выполнить программу, написанную на таком языке, необходимо использование компилятора, то есть программного средства, которое осуществляет перевод исходного текста программ в форму, которая может быть исполнена машиной.

Трудно переоценить важность тестирования компиляторов, ведь ошибка в компиляторе может привести к ошибкам в программах, которые компилируются с его помощью. Более того, подобные ошибки очень трудно диагностировать.

Синтаксический анализ является одной из важнейший фаз компиляции. На данном этапе происходит проверка синтаксической структуры программы и построение синтаксического дерева. Наличие ошибок на данной стадии приводит к некорректной работе всех последующих фаз компиляции, таких как семантический анализ, оптимизация и генерация кода.

Основной целью данной курсовой работы является исследование алгоритма порождения тестов для синтаксических анализаторов, который основан на обобщённом моделировании LL(1)-анализатора.

1. Выбор подхода к тестированию синтаксического анализатора. Разработка грамматики входного языка

Тестирование программного обеспечения — процесс выполнения программы с целью обнаружения ошибок. Программа считается правильно работающей, если:

- Она выдает ожидаемый результат на всех корректных входных данных.
- Она выдает ошибку на всех некорректных входных данных.

Тесты, проверяющие первую ситуацию, называются позитивными, вторую — негативными. Применительно к синтаксическим анализаторам позитивный тест — это последовательность токенов, являющаяся предложением целевого языка, а негативный — это последовательность токенов, не являющаяся предложением целевого языка. Считается, что позитивный тест должен быть построен так, чтобы при получении такого теста на вход использовалось как можно больше путей выполнения в программе. В то же время для негативного тест должен содержать только одну ошибку [1]. Приминительно к компиляторам позитивный тест должен содержать цепочки, для вывода которых используется максимальное количество правил грамматики, а негативный должен быть спроектирован так, чтобы при раскрытии какого-нибудь одного нетерминала происходила ошибка.

Существует два основных подхода к тестированию программного обеспечения: метод черного ящика, который предполагает отсутствие информации о внутренней структуре программы, и метод белого ящика, предполагающий наличие такой информации. В данной работе рассматривается тестирование методом белого ящика. При данном подходе эффективность работы тестера измеряется степенью покрытия логики программы. Часто покрыть всю логику программы не представляется возможным, поэтому выработка критерия полноты покрытия является необходимым условием реализации алгоритма, который генерирует тесты.

1.1. Генерация позитивных тестов

В конце 60-х годов появляются первые подходы к написанию тестов по грамматике. Изначально это были стохастические алгоритмы, пораждающие по грамматике всевозможные предложения языка [2]. Несмотря на то, что данный подход позволяет генерировать сколь угодно большое количество тестов, возможность его использования на практике вызывает большие сомнения. Во-первых, большинство из этих тестов будут дублировать друг друга. Во-вторых, нет возможности покрыть все пути выполнения. В-третьих, отсутствует критерий остановки. Все эти факторы приводят к тому, что невозможно предсказать время, за которое компилятор будет протестирован.

Впервые критерий полноты покрытия для позитивных тестов был сформулирован Пардомом [3]. В соответствии с его критерием полное покрытие логики программы достигается в случае, если для каждого правила грамматики существует тест, в котором для вывода предложения языка используется это правило.

Существуют более продвинутые критерии полноты покрытия. Например, в качестве такого критерия можно взять критерий покрытия всех пар [4]. Пусть N — множество нетерминалов грамматики, T — множество терминалов, тогда пара (A, t), где $A \in N, t \in T$ считается покрытой, если в процессе работы LL-распознавателя возникает ситуация, когда на вершине стека находится символ A, а текущим входным символом является t.

Для того, чтобы получить набор позитивных тестов, удовлетворяющих данному критерию, сначала нужно для каждого нетерминала грамматики (с аксиомой S) A построить следующую цепочку:

$$A \prec B_k \prec ... \prec B_1 \prec S,$$

где все B_i различны, а отношение $A \prec B$ означает, что в грамматике существует правило, в котором в левой части стоит нетерминал B, а в правой части одим из символов является A. В процессе построения таких цепочек для каждого терминала получим сентенциальную форму вида $\alpha A \beta$, где $\alpha, \beta \in (N \cup T)^*$.

Определим функцию $\mathcal{F}(A)$, где $A \in (N \cup T)^*$, возвращающую множество раскрытий A, т.е множество цепочек, выводимых из A путем замены первого нетерминала до тех пор, пока первым символом не будет терминал:

- Если A — терминал, то $\mathcal{F}(A) = A$
- Если A — нетерминал, то

$$\mathcal{F}(A) = \bigcup_{p=A\to\alpha} \mathcal{F}(\alpha)$$

- Если $A = A_1 \dots A_n$ — сентенциальная форма, то

$$\begin{cases} \mathcal{F}(A) = \{\beta A_2 \dots A_n \mid \beta \in \mathcal{F}(A_1)\}, \text{ если } \epsilon \notin \mathcal{F}(A_1) \\ \mathcal{F}(A) = \{\beta A_2 \dots A_n \mid \beta \in \mathcal{F}(A_1)\} \cup \mathcal{F}(A_2 \dots A_n), \text{ иначе} \end{cases}$$

В итоге получаем для каждого нетерминала сентенциальную форму вида $\alpha A \beta$ и множество $\mathcal{F}(A)$. Это позволяет получить множество цепочек вида $\alpha\gamma\beta$, где $\gamma \in \mathcal{F}(A)$. Для каждой такой цепочки можно вывести предложение языка. Множество таких предложений для всех нетерминалов языка удовлетворяет критерию покрытия пар.

1.2. Генерация негативных тестов

Все предыдущие подходы описывали процесс генерации позитивных тестов, однако для полноценного тестирования необходимо также сгенерировать и негативные тесты. В настоящее время существует не так много подходов к решению данной проблемы.

Наиболее популярным методом для порождения негативных тестов является метод мутаций. Суть данного подхода заключается в изменении предложения языка двумя различными способами: вставкой произвольного терминала и заменой произвольного терминала на другой терминал. Применение этого метода без каких-либо модификаций влечет за собой проблему, которая заключается в том, что после применения мутации может получиться предложение языка. У этой проблемы существует два решения. Во-первых, можно использовать эталонный компилятор для того, чтобы определить, является ли измененная цепочка предложением языка. Очевидный недостаток такого метода заключается в том, что "эталонный" компилятор далеко не всегда существует. Во-вторых, можно вставлять не произвольный терминал, а только тот, который не может находится в данном позиции. Аналогично решается проблема с заменой.

Второй способ генерации негативных тестов также заключается в применении метода мутаций, но уже к грамматике языка: в грамматику вносятся изменения (мутации) для получения грамматик, порождающих близкие, но не эквивалентные исходному языки [4]. Затем эти грамматики подаются на вход генератору тестов для получения потенциально негативных тестов. Проблемы данного способа также заключаются в возможном порождении эквивалентных грамматик и позитивных тестов вместо негативных. В данном случае решить эти проблемы можно только с использованием эталонного компилятора.

1.3. Комбинированный подход

Так как ни один из описанных выше алгоритмов полностью не удовлетворяет обозначенным критериям, то было решено использовать их комбинацию. В качестве основы был выбран алгоритм предсказывающего синтаксического разбора. Данный алгоритм определяет, является ли входная цепочка предложением языка, и, если является, то порождает дерево синтаксического разбора. В процессе работы такой алгоритм, работая с входной цепочкой, использует вспомогательные структуры данных: магазин, который представляет собой стек, содержащий терминалы, нетерминалы и концевой маркер, и таблицу анализатора, которая позволяет по нетерминалу со стека и терминалу из входной цепочки определить, по какому правилу раскрывать нетерминал. В случае, если раскрытие невозможно, соответствующая ячейка таблицы содержит специальный объект, означающий ошибку.

Протокол работы алгоритма предсказывающего анализа заключается в рассмотрении символа X на вершине стека и текущего входного символа a. Всего имеется три возможности:

1) Если $X = a = \$$, где $\$$ — концевой маркер, то цепочка успешно разобрана.

2) Если $X = a \neq \$$, то X удаляется из стека и указатель входа передвигается на следующий символ.

3) Если X — нетерминал, то необходимо обратиться в таблицу разбора по паре (X, a), если в этой ячейке находится "ошибка", то генерируем сообщение об ошибке, иначе снимаем со стека X и помещаем на стек содержимое ячейки.

Для адаптации оригинального алгоритма было решено использовать недетерминистские вычисления: при работе алгоритма встречаются точки ветвления, в этих точках вычислительный процесс клонируется и в каждом дочернем процессе вычисление идет по своему пути. После окончания работы алгоритма собираются результаты работы всего дерева дочерних процессов. Для моделирования такого подхода на детерменированной машине в точке ветвления процесс продолжает выполнение по первому пути, а состояния вычислительной среды для остальных путей кладутся в стек. После завершения одного из процессов происходит извлечения очередного состояния из стека, и выполнение продолжается. Данные действия выполняются, пока стек не пуст.

Состоянием вычислительной среды для генератора тестов является префикс сгенерированной цепочки, магазин, следующий символ и булевский флаг, означающий тип генерируемого теста ("позитивный/негативный"). При работе распознавателя в точке, где происходит переход к следующему символу, мы рассматриваем все допустимые входные символы для "позитивных" тестов и недопустимые для "негативных" тестов. В качестве критерия полноты покрытия был выбран критерий покрытия всех пар, описанный ранее.

Для конкретной реализации алгоритма (Алгоритм 1) определяются два состояния — "открытое" и "закрытое". В "открытом" состоянии следующий символ может быть произвольным, а в "закрытом" следующий символ известен. Когда мы переходим из "открытого" состояния в "закрытое", то происходит запись символа в выходную цепочку. Обратный переход заключается в "потреблении" терминала. Также в алгоритме используется специальная функция "fork", которая моделирует клонирование процесса.

Данный алгоритм (Алгоритм 1) описывает процесс генерации позитивных тестов, удовлетворяющих описанным критериям, для генерации негативных тестов методом мутаций необходима модификация. В открытом состоянии, когда на стеке нетерминал X, для моделирования вставки необходимо до добавления в цепочку терминала a добавить терминал b такой, что ячейка таблицы разбора, соответствующая паре (X, b) не была посещена и содержала "ошибку". Для моделирования замены надо вместо добавления в выходную цепочку терминала a добавить в выходную цепочку символ b и перейти в "закрытое" состояние с a на входе. После перехода "закрытое" состояние и в том и в другом случае

необходимо сгенерировать кратчайшую цепочку, т.к. негативный тест должен содержать лишь одну ошибку и быть минимальным по размеру.

Необходимо также отметить, каким образом решается проблема остановки генерации тестов. Для этого в "открытом" состоянии при наличии нетерминала на стеке выбираются только такие терминалы a, для которых пара (N, a) еще не была посещена. Когда при работе в магазине остается только концевой маркер, то происходит запись тестов либо в память, либо в файл, и работа заврешается.

Множество $FIRST(u)$, которое используется в алгоритме, определяется как множество терминалов, с которых начинаются цепочки, выводимые из u. Кроме того, нужно описать процесс построения таблицы предсказывающего анализатора, которая используется в алгоритме для определения момента, в который нужно прекратить генерировать тесты (критерий остановки). Для того, чтобы построить таблицу предсказывающего анализатора, помимо множества $FIRST(u)$, определенного ранее, требуется множество $FOLLOW(X)$ — множество терминалов a таких, что существует вывод вида $S \Rightarrow^{\star} uXav$, где $X \in N$, S — аксиома грамматики, u, v — цепочки символов. Тогда построить таблицу можно следующим образом:

1) Положим для каждой ячейки значение $ERROR$.

2) Для каждого правила грамматики $X \to u$:

   а) Для каждого $a \in FIRST(u)$ добавляем u к ячейке, соответствующей паре (X, a).

   б) Если $\epsilon \in FIRST(u)$, то для каждого $b \in FOLLOW(X)$ добавляем u к ячейке, соответствующей паре (X, b).

Алгоритм 1 Порождение позитивных и негативных тестов по грамматике

function TestsGenerator

1. **"Закрытое" состояние (следующий символ известен)**:
   - на стеке $x \in T$ — снимаем символ → "открытое"
   - на стеке $x \in N$ — раскрываем нетерминал → "закрытое".
2. **"Открытое" состояние (следующий символ любой)**:
   - на стеке $x \in T$ — снимаем символ → "открытое"
   - на стеке $x \in N$:
     Если $\epsilon \notin FIRST(x)$:
       Для всех $a \in FIRST(x)$ ((x, a) еще не посещена):
         - делаем fork(a)
         - добавляем в выходную цепочку a
         - переходим в "закрытое" состояние с a на входе

     Иначе:
       Для всех $a \in FIRST(x) \setminus \{\epsilon\}$ ((x, a) еще не посещена):
         - делаем fork(a)
         - добавляем в выходную цепочку a
         - переходим в "закрытое" состояние с a на входе

       Для $\epsilon \in FIRST(x)$
         - снимаем нетерминал → "закрытое"

end function

1.4. Разработка грамматики входного языка

Чаще всего спецификация языка задается с помощью контекстно-свободной грамматики $G = \langle N, T, P, S \rangle$, где N — множество нетерминалов грамматки, T — множество терминалов грамматки, P — множество правил вывода, каждое из которых имеет вид $A \to \gamma$, где $A \in N, \gamma \in (N \cup T)^*$, а $S \in N$ — аксиома языка. Правила вывода удобно записывать с помощью РБНФ — расширенной формы Бэкуса-Наура, которая позволяет для записи правил использовать символы: '|' — альтернатива, '*' — вхождение 0 или более раз, '+' — вхождение 1 или более раз, '?' — вхождение 0 или 1 раз. Чтобы позволить пользователю вводить описние языка с помощью РБНФ, необходимо описать грамматику входного языка:

⟨*sample*⟩ ::= ⟨*header*⟩ ⟨*main*⟩

⟨*header*⟩ ::= 'non-terminal' ident (',' ident)* ';' 'terminal' ident (',' ident)* ';'

⟨*main*⟩ ::= ⟨*rule*⟩+

⟨*rule*⟩ ::= ident '::=' ⟨*item*⟩+ ';'

$\langle item \rangle$ ::= ident+
| $\langle item \rangle$ ('|' $\langle item \rangle$)+
| '(' $\langle item \rangle$+ ')'
| $\langle item \rangle$ ('*' | '+' | '?')

В данной записи нетерминалы обозначаются с помощью треугольных скобок, терминалы — с помощью кавычек. С помощью слова "ident" обозначаются идентификаторы — последовательности букв и цифр, которые начинаются с буквы.

2. Реализация лексического и синтаксического анализаторов

Для того, чтобы выделить из грамматики, которую ввел пользователь, необходимую для алгоритма информацию, нужно реализовать лексический и синтаксический анализаторы.

В качестве языка программирования для написания тестера синтаксических анализаторов был выбран язык Python. Данный язык программирования нацелен на уменьшение времени разработки сложных программных продуктов, а также имеет простой и удобный синтаксис, что позволяет существенной сократить время реализации. Более того, существует множество библиотек для языка Python, позволяющих по описанию грамматики сгенерировать как лексический, так и синтаксический анализаторы.

В качестве программного средства для генерации лексического и синтаксического анализаторов был выбран ANTLR [5] (ANother Tool for Language Recognition). ANTLR позволяет по грамматике, записанной в РБНФ (листинг 1), породить лексер и парсер, которые затем можно использовать в программе. Лексер позволяет выделить из входного текста поток лексем, а парсер, получая на вход поток лексем, возвращает синтаксическое дерево. ANTLR предоставляет средства для обхода данного дерева. После получения ANTLR дерева разумно выполнить следующие действия:

- Провести семантический анализ — проверку того, что все используемые терминалы и нетерминалы определены. В случае, если какой-либо используемый символ не был определен, порождается исключение, и программа выдает сообщение об ошибке.

- Осуществить переход от конкретного синтаксического дерева, которое строит ANTLR к абстрактному, т.е. выделить терминалы, нетерминалы и правила вывода.

Листинг 1. РБНФ грамматика для ANTLR

```
grammar InputGrammar;

sample
    : header main
    ;

header
    : KW_NT IDENT (OP_COM IDENT)* OP_SC KW_T IDENT (OP_COM IDENT)*
       OP_SC
    ;

main
    : grammar_rule+
    ;

grammar_rule
    : IDENT OP_EQ complex_item OP_SC
    ;

complex_item
    : item+ | item+ (OP_OR item+)+
    ;

item
    : (IDENT | EPS) | OP_LP item (OP_OR item+)+ OP_RP | OP_LP item+
        OP_RP | item (OP_MUL | OP_PLUS | OP_QUEST)
    ;
```

Основной структурой данных для описания абстрактного синтаксичекого дерева является класс Symbol, который описывает представление символа входного языка. Данный класс содержит два основных экземплярных поля: поле symbol_type, которое определяет тип символа (терминал, нетерминал, служебный символ или ϵ) и поле image, которое хранит строковое представление символа. Класс EBNFStructure служит для описания структуры РБНФ (нетерминала item из листинга 1). Данный класс также содержит экземплярное поле, обозначающее тип структуры, т.е. какая альтернатива была выбрана в правиле для item, и массив ссылок на дочерние структуры РБНФ или символы. Класс

Rule представляет правило входного языка. Каждое правило состоит из левой части (экземпляр класса Symbol) и правой части (массив экземпляров класса EBNFStructure). Данный класс соответствует классу grammar_rule из листинга 1.

Наконец, для того, чтобы трансформировать синтаксическое дерево, порожденное парсером ANTLR, используется класс ASTParser. В конструктор данного класса передается ANTLR дерево. Далее происходит парсинг части дерева, которая соответствует нетерминалу header из грамматики для ANTLR. Из этой части выделяются терминалы и нетерминалы. Для каждого такого символа создается символа создается экземпляр класса Symbol. После этого из поддерева, которое соответствует нетерминалу main, выделяются правила грамматики, записанные в РБНФ. Для каждого из этих правил создается экземпляр класса Rule.

3. Реализация генератора тестов

Во-первых, так как пользователь может вводить грамматику входного языка, используя РБНФ, а в процессе работы алгоритма порождения тестов необходимо наличие правил в формате БНФ, то необходимо перевести правила, записанные в РБНФ в БНФ. Этого можно добиться следующим образом:

- Каждое правило вида $x \to \ldots y^{\star} \ldots$ заменяем на $x \to \ldots x_1 \ldots, x_1 \to \epsilon | y x_1$.
- Каждое правило вида $x \to \ldots\ y^{+} \ldots$ заменяем на $x \to \ldots\ y\, x_1 \ldots,\ x_1 \to \epsilon | y x_1$.
- Каждое правило вида $x \to \ldots\ y^{?} \ldots$ заменяем на $x \to \ldots\ x_1 \ldots,\ x_1 \to \epsilon | y$.

- Каждое правило вида $x \to \ldots (a|b) \ldots$ заменяем на $x \to \ldots x_1 \ldots, x_1 \to a|b$.

После преобразования РБНФ в БНФ имеет смысл удаление бесполезных символов, то есть символов, которые не могут участвовать в выводе. Наличие таких символов может означать, что пользователь допустил ошибку при описании грамматики входного языка. Более того, удобнее работать с грамматикой, у которой все символы являются полезными. Данный алгоритм реализуется в два шага [6]:

1) Удаление непорождающих нетерминалов (т.е. таких, из которых не может быть выведена терминальная цепочка).

2) Удаление недостижимых нетерминалов (т.е. таких, которые не стоят в правых частях никакого правила и не являются аксиомой грамматики).

Чтобы удалить непорождающие нетерминалы, необходимо [6]:

1) Множество порождающих нетерминалов пусто.

2) Находим правила, не содержащие нетерминалов в правых частях и добавляем нетерминалы, встречающихся в левых частях таких правил, в множество.

3) Для всех правил, где в правой части все нетерминалы уже входят во множество порождающих нетерминалов, левую часть правила добавляем в множество.

4) Повторяем предыдущий шаг, пока множество порождающих нетерминалов изменяется.

Все нетерминалы, не вошедшие в множество порождающих, являются непорождающими.

Чтобы удалить недостижмые нетерминалы, необходимо [6]:

1) Множество достижимых нетерминалов включает в себя только аксиому грамматики.

2) Если в левой части правила стоит элемент из множества достижимых нетерминалов, то добавляем все нетерминалы из правой части в множество.

3) Повторяем предыдущий шаг, пока множество достижимых нетерминалов изменяется.

Все нетерминалы, не вошедшие в множество достижимых, являются недостижимыми.

Во-вторых, в процессе реализации алгоритма стало очевидно, что при порождении негативных тестов после вставки некорректного символа необходимо раскрыть нетерминалы на стеке так, чтобы получилось кратчайшая цепочка (т.к. негативные тесты должны быть как можно короче). Для того, чтобы в процессе программы не приходилось каждый раз высчитывать кратчайшую цепочку, было принято решение перед запуском процедуры порождения тестов построить кратчайшую цепочку для каждого нетерминала и сохранить их в словарь. Сам по себе алгоритм получения кратчайшей цепочки для нетерминала выглядит следующим образом. Пусть $\alpha, \gamma, \beta \in T^*, X, Y \in N, X \to \alpha$, тогда:

1) Если $|\alpha| = 0, \alpha = \epsilon$, то заменяем все правила вида $Y \to \gamma X \beta$ на $Y \to \gamma\beta$.

2) Если $|\alpha| = 1$, то заменяем все правила вида $Y \to \gamma X \beta$ на $Y \to \gamma\alpha\beta$.

...

3) Если $|\alpha| = n$, то заменяем все правила вида $Y \to \gamma X \beta$ на $Y \to \gamma\alpha\beta$.

Выполняем эти шаги до тех пор, пока в правых частях всех правил не останутся только терминалы. Это гарантированно случится, так как мы удалили все бесполезные нетерминалы.

После выполнения подготовительных действий, описанных выше, можно перейти к реализации самого тестера. Для описания состояния вычислительной среды используется класс State. Конструктор данного класса получает на вход префикс сгенерированной цепочки, текущее состояние магазина, тип теста (позитивный или негативный) и следующий входной символ. Также данный класс содержит вспомогательные методы для получения и удаления последнего символа со стека состояния, раскрытия последнего правила на стеке. Класс Tester отвечает непосредственно за порождение тестов. Его конструктору на вход подается объект класса ASTParser, через который осуществляется доступ к абстрактному синтаксическому дереву, и путь, по которому необходимо записать тесты. Tester создает папку "tests", внутри которой содержатся папки "negative" и "positive" для негативных и позитивных тестов соответственно. Объект данного класса строит множества FIRST и FOLLOW, таблицу предсказывающего анализатора и затем порождает тесты по алгоритму, описанному выше. При этом каждый тест записывается в отдельный файл.

4. Тестирование

Для тестирования на вход подавались различные грамматики. В частности: грамматика формата JSON (JavaScript Object Notation), арифметических выражений, подмножества языка Pascal. Одним из основных критереев удобства работы с программой является ее скорость.

Название	Нетерминалов	Терминалов	Правил	Время (с)
Арифметические выражения	6	7	6	0.14
JSON	5	11	5	0.11
Подмножество Pascal	55	54	55	1.93

Таблица 1. Скорость работы программы

Оценивая скорость работы программы на разных входных данных (Таблица 1), можно сделать вывод о приемлимых результатах, ведь достаточно всего один раз сгенерировать тесты для полноценного тестирования синтаксического анализатора. Поэтому секунды (и даже десятки секунд) являются удовлетворительным временем работы.

Помимо скорости работы интерес представляет количество тестов, порожденных программой.

Название	Позитивных	Негативных	Всего
Арифметические выражения	2	19	21
JSON	5	45	50
Подмножество Pascal	50	2114	2164

Таблица 2. Количество тестов

Как видно из таблицы 2 число позитивных тестов относительно невелико, что удовлетворяет поставленной задаче. Количество негативных тестов в разы превышает количество позитивных, но в случае автоматизированного тестирования это не должно стать критической проблемой.

Для того, чтобы проверить корректность работы программы, был проведен тест на самоприменимость. Для этого грамматика, которая подавалась на вход antlr, была переписана в виде, которые принимает на вход тестер. В результате

был получен набор позитивных и негативных тестов, позволяющий протестировать саму программу порождения тестов. Для этого в программу был добавлен специальный режим "grammar-check". В данном режиме программа осуществляет лексический и синтаксический анализ, а также выполняет преобразование синтаксического дерева, которое строит antlr, в набор терминалов, нетерминалов и правил. В случае, если в процессе парсинга произошли ошибки, программа завершается с ненулевым кодом возврата. Для того, чтобы автоматизировать тестирование был написан bash-скрипт, который запускает тестер, подавая на вход сгенерированные тесты, и проверяет код возврата.

В результате работы скрипта выяснилось, что не на всех негативных тестах программа возвращает ненулевой код возврата. Оказалось, что не все "негативные" тесты, порожденные тестером, действительно являются негативными. Причина такого поведения заключается в следующем: если при обработке очередного нетерминала в открытом состоянии все ячейки таблицы, которые соответствуют данному нетерминалу, уже были посещены, то генерируется кратчайшая цепочка. В процессе генерации цепочки возможно раскрытие нетерминала как ϵ (пустое раскрытие). Если такой нетерминал стоял в конце цепочки, то при порождении негативного теста (путем вставки некорректного символа b) можно сгенерировать позитивный тест. Это может произойти если нетерминал, раскрытый как ϵ, мог раскрываться в цепочку, начинающуюся с b. Чтобы решить данную проблему, можно использовать верификацию негативных тестов, с помощью таблицы разбора, построенной в процессе работы алгоритма.

Листинг 2. Пример позитивного теста

```
program ident;
 var
  ident: char;
  ident, ident: boolean;
 var ident: char;
 const
   ident = 'string';
   ident = 'string';
 func: char;
  const ident = 'string';
  procedure proc;
    func (ident: char; ident: char) : boolean;
```

```
      procedure proc (ident : char);
       begin
         ident := ident
       end;
      begin
        ident := ident
      end;
    begin
      ident := ident
    end;
  begin
   ident := ident
  end;
begin
 ident := ident
end.
```

Анализируя позитивный тест (Листинг 2), можно отметить, что данная цепочка является предложением языка Pascal и должна быть корректно обработана синтаксическим анализатором.

Листинг 3. Пример негативного теста

```
program ident;
begin <>
 ident := ident
end.
```

Негативный тест (Листинг 3) представляет собой позитивный тест либо со вставленным некорректным символом (т.е. таким, который не может находиться в месте вставки), либо с заменой корректного символа на некорректный. В данном примере имеет место вставка символа "<>". Очевидно, что эта последовательность терминалов не является предложением языка и может быть использована в качестве негативного теста.

5. Заключение

В рамках данной работы было проведено исследование алгоритма для порождения тестов, основанного на обобщённом моделировании LL(1)-анализатора. Было выяснено, что данный алгоритм успешно справляется с порождением позитивных тестов. Однако, существуют проблемы с генерацией негативных тестов.

Помимо этого существует ряд доработок, которые могли бы улучшить программу и сделать ее более комфортной в использовании. Во-первых, можно изменить входную грамматику так, чтобы пользователю не приходилось в явном виде описывать все терминалы и нетерминалы перед списком правил, вместо этого можно использовать соглашение о том, что все терминалы должны быть заключены в кавычки, а нетерминалы - нет. Это бы позволило упростить ввод грамматики. Во-вторых, можно модифицировать данный алгоритм таким образом, чтобы он мог работать со всеми видами грамматик (тогда в ячейке таблицы может содержаться не одно, а несколько правил).

Список литературы

[1] Майерс Г. Искусство тестирования программ/ Г. Майерс, Т. Баджетт, К. Сандлер. - 3-е издание, переработанное и дополненное - Москва: И.Д. Вильямс, 2012 - 272с.

[2] Hanford K.V., "Automatic generation of test cases," IBM Systems Journal, Vol. 9, No. 4, 1970. pp. 242 - 257.

[3] Purdom P., "A sentence generator for testing parsers," BIT Numerical Mathematics, 1972. pp. 366-375.

[4] Зеленов С.В., Зеленова С.А. Автоматическая генерация позитивных и негативных тестов для тестирования фазы синтаксического анализа // Труды Института системного программирования РАН. 2004. Т. 8.

[5] Домашняя страница проекта ANTLR, URL: http://www.antlr.org/ (дата обращения 26.10.2016).

[6] Опалева Э. А., Самойленко В. П. Языки программирования и методы трансляции. — СПб.: БХВ-Петербург, 2005. — 480 с.