Министерство образования и науки 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)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Генерация тестов для синтаксического анализа методами суперкомпиляции»

Студент ИУ9-82		С. М. Головань
(Группа)	(Подпись, дата)	(И.О.Фамилия)
Руководитель ВКР		А. В. Коновалов
	(Подпись, дата)	(И.О.Фамилия)
Консультант		
	(Подпись, дата)	(И.О.Фамилия)
Консультант		
	(Подпись, дата)	(И.О.Фамилия)
Нормоконтролер		
	(Подпись, дата)	(И.О.Фамилия)

2018 г.

АННОТАЦИЯ

Темой данной работы является «Генерация тестов для синтаксического анализа методами суперкомпиляции». Объем работы составляет 50 страниц.

Основным исследуемым объектом данной работы являются $LL(1)$-грамматики, а точнее – предсказывающий анализатор, распознающий такого рода грамматики. Предметом работы в таком случае является генерация на базе методов суперкомпиляции наборов тестов – позитивных и негативных – с помощью которых возможно исследовать синтаксический анализатор на наличие ошибок в логике его работы.

В дипломную работу входят три главы. Первая из них посвящена теоретическим основам тестирования синтаксических анализаторов и методам суперкомпиляции. Глава «Разработка» посвящена описанию используемых алгоритмов построения и обхода графа конфигураций, более узкой постановке задачи тестирования и определению специфичных для конкретного типа грамматик правил. В главе «Тестирование» рассматриваются различные входные грамматики и результаты работы на них построителя графа конфигураций и генератора тестов.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С развитием языков высокого уровня значение компиляторов и интерпретаторов приобрело промышленные масштабы. При разработке ПО программист рассчитывает, что программа, написанная на исходном языке A будет корректно переведена на язык B.

Именно поэтому тестирование компиляторов занимает важное место в процессе разработки и применения рядовыми пользователями. От каждого компилятора ожидается, что в процессе его работы логика программы не будет искажена, а компилятор при любых допустимых входных данных – возможно, ошибочных с точки зрения грамматики входного языка – либо успешно переведет программу на язык B, либо выдаст сообщение об ошибке.

В данной работе рассматриваются методы генерации тестов для синтаксического анализатора, являющегося важной составляющей стадии анализа.

Теоретическая часть работы опирается на идею суперкомпиляции, предложенную советским ученым В. Ф. Турчиным еще в 70-х годах прошлого века. К сожалению, широкого распространения суперкомпиляция не получила по сей день, но интересна и имеет потенциал в рамках решения определенного ряда задач.

В качестве одной из таких задач рассматривается суперкомпиляция $LL(1)$-грамматик. Преимуществом таких грамматик является простота и интуитивная понятность, а также возможность построения таблицы предсказывающего анализа для выполнения разбора входной цепочки детерминированным распознавателем – автоматом с магазинной памятью, построение графа состояний которого связано с построением графа конфигураций в контексте методов суперкомпиляции.

Такая таблица может быть построена «на глаз» с учетом множеств $FIRST$ и $FOLLOW$, построение которых также интуитивно понятно. Состояние магазина

в данном случае описывается некоторым набором нетерминальных и терминальных символов, что в свою очередь позволяет построить наглядное графическое представление, упрощает разработку и тестирование алгоритмов построения графа конфигураций и его обхода.

При этом, например, для LR-грамматики построение таблицы уже существенно сложнее в силу неочевидности находящихся на стеке элементов – состояний распознавателя, а также операций типа ACTION, GO.

Таким образом, целью данной работы является изучение и практическое применение методов суперкомпиляции для построения конечного набора тестов – цепочек терминальных символов грамматики – определенного вида (позитивных и негативных), проверяющих работу синтаксического анализатора, распознающего $LL(1)$-грамматики.

1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 СИНТАКСИЧЕСКИЙ АНАЛИЗ

1.1.1 Постановка задачи синтаксического анализа

Синтаксический анализ – фаза компиляции, группирующая лексемы, порождаемые на фазе лексического анализа, в синтаксические структуры для некоторой заданной порождающей грамматики.

Как было сказано ранее, в данной работе рассматривается класс $LL(1)$-грамматик, поэтому необходимо сказать несколько слов о КС-грамматиках, подмножеством которых и являются $LL(1)$.

Контекстно свободная (КС) грамматика G – кортеж $< N, T, P, S >$, где:

N – множество нетерминальных символов

T – множество терминальных символов

P – набор правил вывода: $A \rightarrow u$, *где* $A \in N, u \in (N \cup T)^*$

S – стартовое правило вывода (аксиома)

Для такой грамматики всегда можно построить **дерево вывода** – упорядоченное дерево, каждая вершина которого помечена символом из множества $N \cup T \cup \{\varepsilon\}$. Символ ε является специальным и обозначает пустое правило раскрытия нетерминала.

Корнем дерева является аксиома грамматики, внутренние вершины – нетерминалы, листья – терминалы, либо ε. Построение дерева производится согласно правилам вывода грамматики.

Опишем теперь основную задачу синтаксического анализа. Пусть дана КС-грамматика $\boldsymbol{G = < N, T, P, S >}$, а также $\boldsymbol{x \in T^*}$ – входная цепочка. Тогда **основной задачей синтаксического анализа** является определение принадлежности входной цепочки заданной грамматике и, в случае положительного ответа, формирование набора данных, по которому возможно построение дерева вывода. Зачатую сам факт выполнения построения такого дерева не имеет значения, однако важным результатом является построение некоторых структур данных, с помощью которых построение дерева вывода будет являятся возможным.

Часто $LL(1)$-грамматики (как и все контекстно-свободные языки) задаются в расширенной форме Бэкуса-Наура (РБНФ), т.е. имеют вид $X \to R,\ X \in N,\ R -$ дерево, задаваемое следующей грамматикой:

$R ::= \varepsilon \mid X \mid R\ R \mid R\ '|'\ R \mid R'*'|R'+' \mid R'?'$, где

$'|' -$ альтернатива, $' * ' -$ вхождение 0 и более раз, $' + ' -$ вхождение 1 и более раз, $'?' -$ вхождение 0 или 1 раз.

Данная работа не является исключением, поэтому все указанные грамматики для удобства задаются в РБНФ.

1.1.2 LL(1)-грамматики

КС-грамматики представляют собой обширный класс языков. Их наиболее распространенным подклассом являются грамматики вида $\boldsymbol{LL(k)}$, для которых можно построить синтаксический анализатор, работающий за линейное время. Для определения раскрытия нетерминального правила такому анализатору требуется информация о следующих $\boldsymbol{k}$ входных символах.

Наибольший интерес здесь представляет класс $LL(1)$-грамматик, для распознавания которых достаточно иметь информацию только о текущем входном символе [1]. Такие грамматики обладают рядом преимуществ, которые

будут рассмотрены в данной работе. Но прежде опишем указанный класс более формально.

Введем вспомогательные множества $FIRST(X)$ и $FOLLOW(X)$, определяемые для любой КС-грамматики $G = < N, T, P, S >$:

$$\boldsymbol{FIRST(X)} = \{\alpha \in T \mid X \Longrightarrow^* av\} \cup \{\varepsilon \mid X \Longrightarrow^* \varepsilon\}$$

$$\boldsymbol{FOLLOW(X)} = \{\alpha \in T \mid S\$ \Longrightarrow^* uXav\}$$

Говоря простым языком, множество $FIRST(X)$ есть множество терминалов, с которых начинаются цепочки, выводимые из X.

В свою очередь $FOLLOW(X)$ – множество терминалов, следующих за выводом цепочки из X. Данное множество может также содержать терминальный символ \$ – концевой маркер, если X может быть самым правым символом некоторой сентенциальной формы.

Как упоминалось выше, грамматика имеет вид $LL(1)$, если для раскрытия любого из её правил достаточно знать только текущий символ входного потока. В терминах множеств $FIRST(X)$ и $FOLLOW(X)$ данное определение можно переформулировать следующим образом:

КС-грамматика G является $\boldsymbol{LL(1)}$, если для любого правила вывода

$A \Rightarrow u \mid v$ выполняется:

1. $FIRST(u) \cap FIRST(v) = \emptyset$
2. *Если* $v \Rightarrow* \varepsilon$, *то* $FIRST(u) \cap FOLLOW(A) = \emptyset$

Как правило, большинство грамматик языков программирования задаются в общем виде, то есть содержат левую рекурсию и неоднозначности.

Поскольку наличие левой рекурсии и неоднозначностей выбора альтернативы, очевидно, не позволяют исходной грамматике быть $LL(1)$,

предварительно применяются соответствующие процедуры, устраняющие эти недостатки (устранение левой рекурсии, левая факторизация).

КС-грамматика может быть преобразована к эквивалентной, в которой множество правил вывода не содержит переходов по $\varepsilon-$правилам, а также цепных правил и недостижимых символов, что в дальнейшем упрощает процесс разбора.

Таким образом, исходная грамматика обычно предварительно обрабатывается с целью оптимизации разбора и возможности быть поданной на вход некоторому синтаксическому анализатору.

1.1.3 Предсказывающий анализ

Как уже было упомянуто ранее, $LL(1)$-грамматики обладают важным свойством: они могут быть распознаны предсказывающим анализатором – автоматом с магазинной памятью, работа которого основана на предварительном построении таблицы предсказывающего разбора [1].

Детерминированный автомат с магазинной памятью – это набор

$M = < A, B, Q, s, T, z, \pi >$, где

A – входной алфавит,

B – стековый алфавит,

Q – множество состояний автомата,

T – множество заключительных состояний автомата,

s – начальное состояние,

z – маркер дна стека (обычно обозначается как «$»),

$\pi: Q \times A \cup \{\varepsilon\} \times B \rightarrow Q \times B^*$ – функция переходов.

В контексте синтаксического анализа описанной ранее $LL(1)$-грамматики $G = < N, T, P, S >$, входной алфавит совпадает со множеством терминальных символов, дополненных маркером дна стека, $A \in T^* \cup \{z\}$. Стековый алфавит – $B \in (T \cup N)^* \cup \{\$\}$, а функция переходов задается таблицей предсказывающего анализатора, $\pi: N \times (T \cup \{\$\}) \rightarrow (N \cup T)^* \cup \{error\}$, где $error$ – индикатор ошибки, $z = \$$ – концевой маркер. То есть, в обычной ячейке такой таблицы располагается некоторый упорядоченный набор (возможно пустой) терминальных и нетерминальных символов, а в ошибочной – индикатор ошибки.

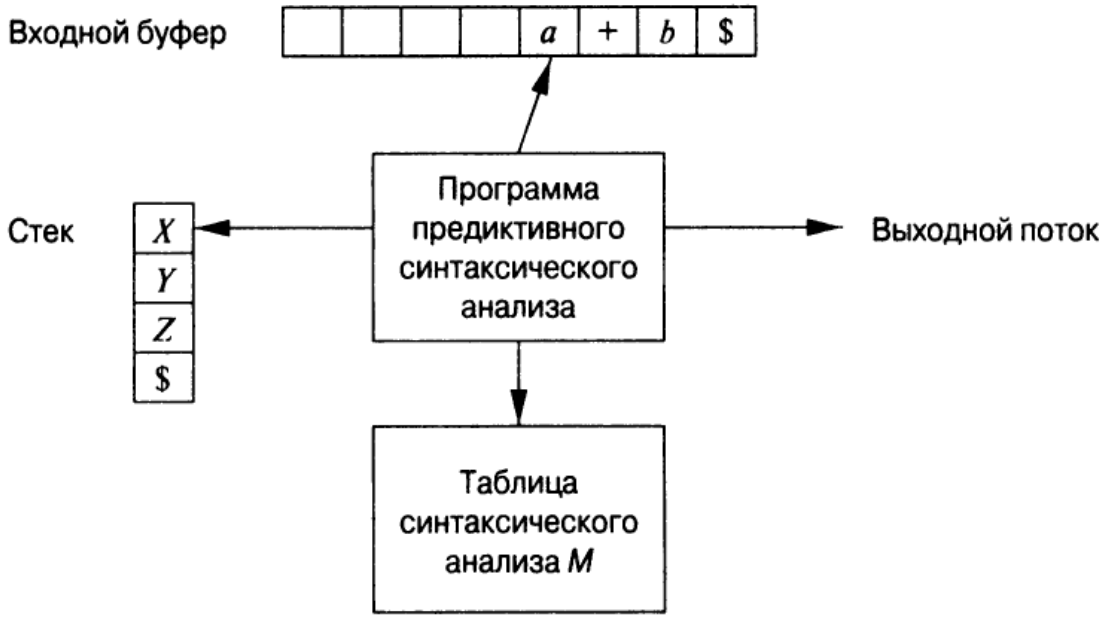

Рисунок 1. Структура предсказывающего анализатора [1].

Стоит отдельно рассмотреть процесс построения таблицы предсказывающего анализатора для $LL(1)$-грамматики. Такая таблица однозначно определяет раскрытие нетерминального правила по текущему терминальному символу входной цепочки. В каждой ячейке таблицы может располагаться единственное возможное раскрытие правила по текущему входному символу или признак ошибки.

	+	*	n	(	)	$
$\boldsymbol{E}$	$error$	$error$	$\boldsymbol{T\,E'}$	$\boldsymbol{T\,E'}$	$error$	$error$
$\boldsymbol{E'}$	$+\,\boldsymbol{T\,E'}$	$error$	$error$	$error$	$\boldsymbol{\varepsilon}$	$\boldsymbol{\varepsilon}$
$\boldsymbol{T}$	$error$	$error$	$\boldsymbol{F\,T'}$	$\boldsymbol{F\,T'}$	$error$	$error$
$\boldsymbol{T'}$	$\boldsymbol{\varepsilon}$	$*\,\boldsymbol{F\,T'}$	$error$	$error$	$\boldsymbol{\varepsilon}$	$\boldsymbol{\varepsilon}$
$\boldsymbol{F}$	$error$	$error$	$\boldsymbol{n}$	$(\,\boldsymbol{E}\,)$	$error$	$error$

Рисунок 2. Таблица предсказывающего анализатора грамматики арифметических выражений

Пусть в начале $\forall X \in N, a \in T \cup \{z\}$: $\pi(X, a) = error$ – признак ошибки. Затем, для всех правил перехода грамматики $X \to u$ выполняем следующее:

1) $\forall a \in FIRST(u): \pi(X, a) \leftarrow u$

2) Если $\varepsilon \in FIRST(u)$, то $\forall b \in FOLLOW(X)$: $\pi(X, b) \leftarrow u$

На Рисунке 2 можно увидеть построенную таблицу предсказывающего разбора для следующей грамматики, называемой грамматикой арифметических выражений:

$E \quad ::= TE'$;

$E' \quad ::= \text{'+'}\ TE' \mid \varepsilon$;

$T \quad ::= FT'$;

$T' \quad ::= \text{'*'}\ FT' \mid \varepsilon$;

$F \ ::= n \mid \text{'('}\ E\ \text{')'}$;

Из такой грамматики, в частности, выводятся цепочки $n * n + n$ и $(n + n)$ для некоторого n. При попытке перейти по ячейке с ошибочным правилом анализатор переходит в режим восстановления, который будет рассмотрен далее.

1.1.4 Восстановление при ошибках

Очевидно, что на вход анализатору не всегда будут подаваться корректные цепочки. В процессе работы предсказывающего анализатора может возникнуть ситуация, когда терминал на верхушке стека не соответствует входному символу или, когда на вершине стека находится нетерминал, а ячейка в таблице, соответствующая данному нетерминалу и текущему входному символу, содержит признак ошибки. В таком случае, интерпретатор может либо остановить разбор цепочки, либо войти в режим восстановления при ошибках, попытавшись заменить или пропустить некорректный терминал на входе, для того, чтобы иметь возможность выявить другие синтаксические ошибки.

Рассмотрим восстановление при ошибках «в режиме паники» [1]. Он основан на пропуске символов входного потока до тех пор, пока не будет обнаружен токен из синхронизирующего множества. Такие множества должны выбираться так, чтобы анализатор мог быстро восстанавливаться после часто встречающихся на практике ошибок.

В качестве синхронизирующих множеств могут быть использованы множества $FOLLOW$ нетерминалов грамматики [1]. Также, если нетерминал может порождать пустую строку, то по умолчанию может быть использована пустая продукция. Если терминал на вершине стека не может быть сопоставлен со входным символом, то терминал снимается со стека и синтаксический анализ продолжается. Тогда, синхронизирующее множество состоит из всех остальных токенов.

	+	*	n	(	)	$
$\boldsymbol{E}$	$error$	$error$	$\boldsymbol{T\ E'}$	$\boldsymbol{T\ E'}$	$\boldsymbol{sync}$	$\boldsymbol{sync}$
$\boldsymbol{E'}$	$+\ \boldsymbol{T\ E'}$	$error$	$error$	$error$	$\boldsymbol{\varepsilon}$	$\boldsymbol{\varepsilon}$
$\boldsymbol{T}$	$\boldsymbol{sync}$	$error$	$\boldsymbol{F\ T'}$	$\boldsymbol{F\ T'}$	$\boldsymbol{sync}$	$\boldsymbol{sync}$
$\boldsymbol{T'}$	$\boldsymbol{\varepsilon}$	$*\ \boldsymbol{F\ T'}$	$error$	$error$	$\boldsymbol{\varepsilon}$	$\boldsymbol{\varepsilon}$
$\boldsymbol{F}$	$\boldsymbol{sync}$	$\boldsymbol{sync}$	$\boldsymbol{n}$	$(\ \boldsymbol{E}\ )$	$\boldsymbol{sync}$	$\boldsymbol{sync}$

Рисунок 3. Таблица предсказывающего разбора для грамматики арифметических выражений, дополненная обычными правилами восстановления

На Рисунке 3 значение в ячейке «synch» означает, что в ней содержится правило восстановления, заданное согласно синхронизирующему множеству для данного нетерминала.

Забегая вперед, следует сказать, что в дальнейшем потребуется полностью заполнить таблицу предсказывающего анализатора, даже для тех терминалов, которые не входят в синхронизирующее множество. При этом необходимо отличать ячейки, в которых находятся правила восстановления от обычных ячеек

таблицы. Описание механизмов заполнения таблицы правилами восстановления описывается в разделе «Разработка».

1.1.5 Граф состояний анализатора

Поскольку предсказывающий анализатор есть автомат с магазинной памятью, для него можно построить некоторый граф (возможно бесконечный). В этом графе вершины являются состояниями магазина, а ребра могут быть двух типов – помеченные символом входной цепочки, если при переходе был потреблен символ, иначе – транзитные, не содержащие атрибутов. Транзитные переходы из каждого состояния строятся согласно альтернативам для нетерминала на верхушке стека. Если на верхушке – терминал, то строится переход в следующее состояние по данному символу.

Выполняя удаление транзитных узлов в графе, а также выделяя циклы к эквивалентным вершинам, можно получить **граф состояний** анализатора. Такой граф также может быть бесконечным, однако отсутствие транзитных переходов и различных вершин с одинаковым состоянием магазина существенно упростит процесс свертки, который будет рассмотрен далее.

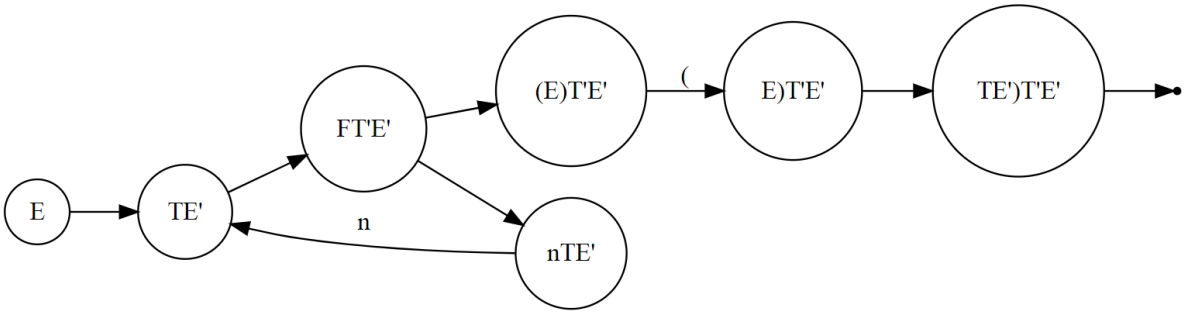

Рисунок 4. Граф переходов грамматики арифметических выражений

На Рисунке 4 продемонстрирован фрагмент графа, который характеризует общий процесс разбора входных цепочек для входной грамматики арифметических выражений. Очевидно, что такой граф является бесконечным, поскольку при переходе по открывающей скобке из состояния $[E]$ в состояние $[E\ )T'E']$, минуя транзитные переходы, дальнейшее развитие верхушки стека повторяется, однако на дне стека всякий раз оказываются новые элементы, что не позволяет выделить циклы к эквивалентным вершинам.

Если дополнить таблицу предсказывающего анализа расширенными правилами восстановления, полностью заполняющими её в ошибочных ячейках, то из каждой вершины графа состояний будут исходить ребра для каждого терминального символа. Под расширенными правилами подразумевается использование обычных правил восстановления с учетом принадлежности текущего входного символа множеству $FOLLOW$ нетерминала на верхушке стека.

В случае отрицательного ответа, для данного символа будет применено несколько другое правило. Формирование таблицы предсказывающего анализатора с учетом расширенных правил восстановления детально описано в разделе «Разработка».

Как станет ясно дальше, именно возможность построения бесконечного графа, построенного с использованием таблицы предсказывающего анализатора и расширенных правил восстановления, а также применение техник суперкомпиляции для сверки такого графа позволяют построить конечный набор тестов, полностью покрывающих всю логику работы синтаксического анализатора.

1.2 ПОСТАНОВКА ЗАДАЧИ ТЕСТИРОВАНИЯ

Получив некоторое представление о предсказывающих синтаксических анализаторах, имеет смысл ввести основные понятия из теории тестирования, а также провести обзор известных на данных момент алгоритмов построения позитивных и негативных тестов для синтаксического анализа.

1.2.1 Основные понятия теории тестирования

Тестирование – процесс исполнения некоторой программы с целью обнаружения в ней ошибок [2].

Таким образом, программа считается правильно работающей, если для всех корректных входных данных она выдает ожидаемый результат, а для некорректных – выдает сообщение об ошибке. При этом важен факт выдачи сообщения об ошибке, а не аварийное (не предусмотренное) завершение работы приложения.

Тесты, которые состоят из корректных входных данных называются **позитивными. Негативными** тестами (тестами на «отказ») называются тесты для некорректных входных данных.

Основной закон построения таких тестовых наборов состоит в том, что при генерации позитивных тестов должно быть покрыто как можно больше различных путей в программе. При этом негативный тест должен содержать единственную ошибку.

Поскольку в большинстве случаев покрыть всю логику работы программы не представляется возможным, появляется необходимость формулировки критерия покрытия тестами, часто называемого также **критерием полноты тестирования.**

Рассмотрим подробнее процесс генерации тестовых наборов для синтаксических анализаторов. Ожидается, что такой набор тестов будет минимальным, то есть будет покрывать всю возможную логику работы парсера при минимально возможной длине цепочки.

Позитивными в таком случае являются тесты, проверяющие работу анализатора, подавая на вход цепочки, являющиеся предложениями языка.

Поскольку корректный синтаксический анализатор не должен аварийно завершать работу на некорректных цепочках, необходимы также ввести негативные тесты, содержащие в себе одну синтаксическую ошибку.

По определению позитивных и негативных тестов ожидается, что набор позитивных тестов будет покрывать все корректные состояния синтаксического анализатора, а набор негативных, соответственно, все ошибочные состояния.

1.2.2 Обзор методов тестирования синтаксических анализаторов

Первые подходы к написанию тестов по грамматике появились еще в конце 60-х годов. В основном это были стохастические алгоритмы [3], порождающие по грамматике все возможные предложения языка – позитивные тесты.

Однако, главной проблемой таких алгоритмов являлось возможное дублирование тестов, а также отсутствие гарантии прохождения всех возможных путей выполнения и критерия остановки.

В 1972 году П. Пардом сформулировал первый критерий полноты для позитивных тестов, по которому полное покрытие логики программы достигается в случае существования для каждого правила грамматики теста, в котором оно используется для вывода предложения языка [4].

Одним из более продвинутых критериев полноты является **критерий покрытия всех пар** [5]. В контексте работы предсказывающего анализатора он формулируется следующим образом:

Пусть $A \in N,\ t \in T$ *– некоторые нетерминал и терминал грамматики, соответственно. Пара* (A, t) *является покрытой, если в процессе работы распознавателя имеется ситуация, когда на вершине стека находится нетерминал* A*, а текущим входным символом является терминал* t*.*

Тогда тестирование синтаксического анализатора является полным в том и только в том случае, если в процессе вывода слов из тестового набора окажутся все возможные пары (A, t) *окажутся покрытыми.*

Данный критерий позволяет строить конечный набор позитивных тестов. Однако для полноценного тестирования необходимо иметь возможность обработать и все некорректные ситуации, то есть построить набор негативных тестов.

Одним из популярных методов генерации негативных тестов является метод мутаций. Он заключается в изменении предложений языка путем применения одной из двух модификаций: вставки произвольного терминала или замены произвольного терминала на другой. Важным моментом здесь является то, что после применения таких модификаций не должно получиться предложение языка. В связи с этим необходимы дополнительные проверки, часто с использованием «эталонного» компилятора, который не всегда существует.

Иногда мутацию применяют не к предложениям, а к самой грамматике языка, получая новую измененную грамматику, которую затем подают на вход генератору тестов с целью получения потенциально негативных тестов.

Проблема данного подхода заключается в возможности порождения позитивных тестов вместо требуемых негативных, а также возможность получения эквивалентных грамматик в результате мутаций.

Обобщая проблемы вышеперечисленных критериев, можно увидеть, что проблемы в построении возникают в силу отсутствия приемлемой модели вычислений в грамматике, используя которую можно было бы формализовать построение как позитивных, так и негативных тестов.

Как станет ясно далее, именно применение методов суперкомпиляции позволяет построить такую модель.

Говоря о тестировании синтаксического анализатора $LL(1)$-грамматики, можно естественно сформулировать такой критерий в первом приближении:

> *Тестирование предсказывающего синтаксического анализатора является полным тогда и только тогда, когда в процессе вывода слов из тестового набора окажутся посещены все ячейки таблицы предсказывающего разбора.*

Данный критерий распространяется на позитивные и негативные наборы тестов. В частности, в процессе построения позитивных тестов должны быть посещены все корректные ячейки таблицы предсказывающего разбора, а при построении негативных – все ошибочные ячейки, содержащие правила восстановления.

Как будет установлено в разделе «Разработка», рассмотренные ранее методы восстановления при ошибках позволят переформулировать такой критерий в терминах графа конфигураций.

1.3 СУПЕРКОМПИЛЯЦИЯ

1.3.1 Основные понятия

Суперкомпиляция (supercompilation) – процесс моделирования выполнения некоторой программы в общем виде с целью её преобразования или анализа. Под словосочетанием «в общем виде» подразумевается построение семантического дерева выполнения исходной программы, учитывающего все возможные входные данные – **дерева конфигураций.** Вершинами такого дерева являются «конфигурации» – обобщенные состояния программы, включающие переменные и вызовы функций.

В общем случае дерево является бесконечным и предпринимается попытка свернуть данное его в конечный **граф конфигураций.** По полученному графу возможно построить «**остаточную**» **программу**, которая в некотором смысле будет являться оптимальной при заданных ограничениях на входные данные [7].

Говоря о суперкомпиляции $LL(1)$-грамматик с целью порождения тестов для синтаксического анализатора, процесс построения остаточной программы – оптимизированной грамматики входного языка – не является приоритетным. Одной из главных задач здесь является построение и свертка бесконечного дерева в конечный граф. Вершинами такого графа будут являться состояния магазина предсказывающего анализатора, а ребрами – переходы по некоторому терминалу в следующее состояние. Далее для выполнения свертки вершины будут разделены на **конфигурации** и $\boldsymbol{let}$**-узлы**, речь о которых пойдет в пункте «Свертка дерева конфигураций».

Выполняя проход по графу конфигураций определенным образом и выписывая атрибуты ребер – терминальные символы – можно построить

необходимые наборы тестов. Процесс построения и обхода графа конфигураций для входной грамматики описан в разделе «Разработка».

1.3.2 Граф конфигураций

Как было сказано ранее, дерево конфигураций задается при выполнении некоторой программы «в общем виде». Поскольку в общем случае программа может содержать циклы и/или рекурсию, дерево может являться бесконечным.

Граф конфигураций – это семантическое дерево программы, над которым произведены следующие преобразования:

1) Удаление транзитных переходов – удаление вершин дерева, являющихся промежуточными, не зависящими от конфигурационных переменных.

2) Выделение эквивалентных вершин – процесс поиска вершины с эквивалентным состоянием, и, в случае нахождения таковой, добавление ребра к найденной конфигурации вместо создания новой.

3) Свертка – процесс выделения отношений между вершинами дерева с целью превращения исходного бесконечного дерева в конечный граф.

Указанные здесь преобразования, как станет ясно позже, гарантируют конечность полученного графа, а значит гарантируют возможность его полного обхода с целью построения тестовых наборов цепочек.

1.3.3 Свертка дерева конфигураций

Рассмотрим методы свертки бесконечного дерева конфигураций. Свертка естественным образом разделяется на два инструмента: **вложение** и **обобщение**. Оба эти инструмента выявляют «сходство» некоторой конфигурации A с конфигурациями-предками, разделяя её на независимые составные части [8].

Вершина графа конфигураций, в которой произошла свертка, обозначается, так называемой, ***let*-вершиной.** Такая вершина имеет только двух потомков: нижнего и верхнего, развитие которых определяется согласно наличию вложения либо обобщения. Также, такая вершина подразумевает независимость её потомков, что означает окончание в ней поиска наличия вложения или обобщения для всех вершин её подграфа. Таким образом, с помощью let-вершин обеспечивается непосредственная свертка дерева в граф.

Вложение представляет собой выделение уже вычисленной ранее части конфигурации, с образованием цикла к найденной подконфигурации (общего собственного префикса двух стеков) и дальнейшим разбором уже оставшейся части конфигурации.

Говоря более формально, если имеется конфигурация со стеком $[Top]$, а среди её предков существует конфигурация $[Top][Context]$, то исходная конфигурация замещается let-вершиной, из которой верхняя дуга исходит к конфигурации конфигурации $[Top]$, а нижняя – к новой конфигурации B, развивающейся, как уже было сказано ранее, независимо.

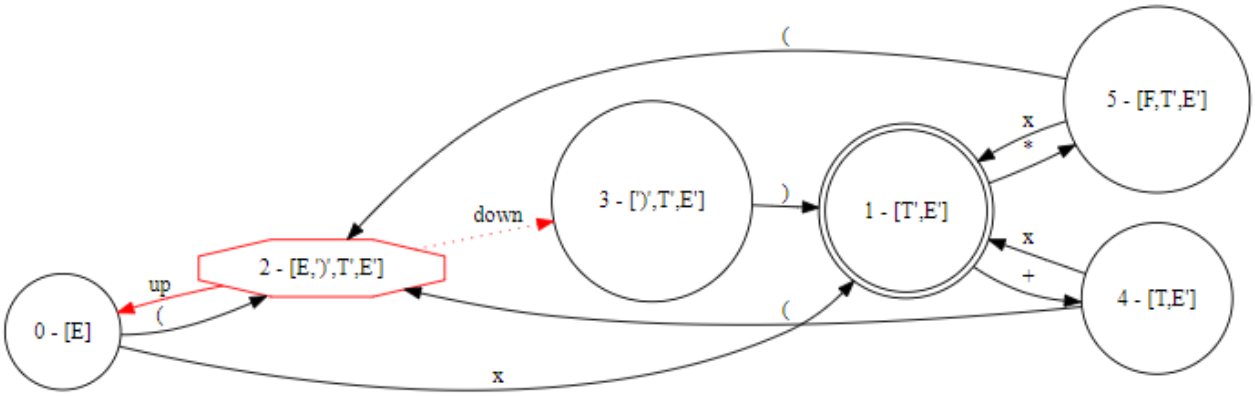

Рисунок 5. Граф конфигураций для грамматики арифметических выражений (без учета ребер восстановления)

На Рисунке 5 приведен граф конфигураций для грамматики арифметических выражений (для наглядности, здесь опущены ошибочные переходы). Многоугольником здесь обозначена let-вершина, характеризующая наличие вложения. Из такой вершины всегда исходят две дуги: верхняя и нижняя, причем при вложении верхняя дуга исходит в корневую конфигурацию.

Однако не всегда возможно обойтись вложением для свертки дерева конфигураций. Как будет рассказано далее, в процессе построения могут возникать ситуации, когда стек начинает разрастаться, при том, что его префикс и суффикс (оба ненулевой длины) остаются неизменными. В связи с этим, далее будут описаны теоретические основы, на которые опирается инструмент обобщения, позволяющий свернуть граф в таких ситуациях.

1.3.4 Отношение Турчина

При решении задачи поиска семантических циклов в программе В. Ф. Турчин рассматривал преобразование стека функций, описываемое **префиксной грамматикой**, в которой также описывается изменение состояний автомата с магазинной памятью [6].

Тройка $G =< A, S_0, R >$, где A – алфавит, $S_0 \in A^+$ – начальное слово (возможно пустое), $R \in A^+ \times A^*$ – конечный набор правил переписывания, в котором правила вывода $R_l \to R_r$ применимы только к словам вида $R_l D$, где D – суффикс, R_l – префикс, называется **префиксной грамматикой.**

Обобщение сводится к поиску семантических циклов в программе с целью исключить ситуации, когда вершина стека проходит один и тот же путь развития неограниченное количество раз, в конце пути повторяя сама себя.

Схематично можно изобразить такую ситуацию следующим образом:

$$fa \to fma \to fmma \to fmmma \to \cdots$$

Или, говоря в контексте состояний анализатора:

$$[Top][Context] \to \cdots \to [Top][Middle][Context] \to \cdots \to$$
$$[Top][Middle]^n[Context].$$

Здесь $[Top]$, $[Context]$ и $[Middle]$ – непустые подмножества в стеке.

Рассматривая цепочки, порожденные префиксной грамматикой, проведем **временное индексирование** – припишем ко всем буквам слов их временной индекс. Иными словами, просматривая цепочку и выписывая посимвольно каждое слово справа налево, пометим каждый символ тем моментом времени, в который он появился в цепочке.

Пусть P, R, A, B, C – слова, буквы в которых размечены временными индексами. Можно определить отношение эквивалентности слов с точностью до временных индексов: $P \approx R \Leftrightarrow |P| = |R| = n,\ \forall i = 1..n: P[i] = R[i]$.

Определим теперь **отношение Турчина** $P \preccurlyeq R$:

$$P \preccurlyeq R \ \Leftrightarrow\ P = AB, |A| > 0, \qquad R = A'CB \ \text{и}\ \ A' \approx A$$

Пары $P,\ R$, находящиеся в отношении Турчина называются **турчинскими.**

Конструкция обобщения неотделима от определения отношения Турчина. Канонически оно рассматривается для префиксных грамматик с учетом временных индексов, а также для обогащенных префиксных грамматик [6].

Здесь и далее будет рассмотрена «безвременная» версия данного отношения. Такое возможно в силу того, что «безвременное» отношение Турчина включает в себя «временное» в качестве подмножества, а в силу того, что турчинские пары встречаются на каждом пути, порождаемом грамматикой, на этом пути встретятся пары, находящиеся и в «безвременном» отношении.

Однако отсутствие учета временных индексов в общем случае для некоторых грамматик может привести к слишком ранней свертке:

$E \quad ::= AB;$

$A \quad ::= 'a';$

$B \quad ::= 'b'\ ACB;$

Для данной грамматики при развертке нетерминала E возникнут стеки AB и ACB, которые будут свернуты по «безвременному отношению» Турчина, но при этом не соответствуют его определению с временными интексами. В данной работе такие грамматики не будут рассматриваться.

2. РАЗРАБОТКА

2.1 ПАРСЕР ГРАММАТИКИ

Основной интерес в данной работе в конечном итоге представляет построение графа конфигураций для заданной входной грамматики и генерация набора тестовых цепочек при обходе такого графа. Для возможности построения графа необходимо иметь инструмент, получающий на вход описание произвольной $LL(1)$-грамматики, упрощающий её в процессе разбора и строящий для неё таблицу предсказывающего разбора, на основе которой и будет построен граф конфигураций.

В качестве отправной точки использовались материалы курсового проекта студента кафедры ИУ-9 Михаила Макарова. В данной работе был реализован front-end компилятора на языке Python 2.7, получающий на вход описание КС-грамматики и, в случае, если она имеет вид $LL(1)$, строящий для неё таблицу предсказывающего анализатора.

Из исходной программы была извлечена стадия анализа с целью передачи результата её работы, в случае успеха, в формате JSON на вход программе построения графа конфигураций. В противном случае на вход подается сообщение об ошибке. Программы построителя графа конфигураций и генератора тестов написаны на языке JavaScript с использованием стандартов ES6, выполняющийся в среде Node.js.

2.2 ПОСТРОИТЕЛЬ ГРАФА КОНФИГУРАЦИЙ

Одной из основных задач работы является построение модели выполнения разбора цепочки грамматики «в общем виде», то есть построение графа конфигураций.

Построение графа конфигураций можно интерпретировать как построение и свертку некоторого бесконечного графа состояний автомата распознавателя. В процессе путешествия по такому графу при попадании в let-вершину создаются виртуальные переходы из финальных вершин в некоторый граф «высокого порядка» с корнем в вершине, соответствующей нижнему ребру текущей let-вершины.

Как было описано в разделах «Синтаксический анализ» и «Суперкомпиляция», построение такого графа производится с использованием таблицы предсказывающего анализа для данной грамматики. Дополнив таблицу правилами восстановления при ошибках, можно получить граф, содержащий ошибочные переходы, по которым в дальнейшем возможно также построение негативных тестов.

В связи с этим, введем следующее правило для ошибочных ячеек. Пусть имеется ошибочная ячейка $\pi(N, a) = error$. Тогда, если $a \in FOLLOW(N)$, то добавляем в ячейку правило $N \to \varepsilon$, иначе добавляем правило $N \to aN$.

Другими словами, если символ на вершине стека принадлежит синхронизирующему множеству, то нетерминал снимается со стека в попытке продолжить анализ, иначе токен из входного потока пропускается.

	+	*	n	(	)	$
$\boldsymbol{E}$	$+\,E$	$*\,E$	$\boldsymbol{T\,E'}$	$\boldsymbol{T\,E'}$	ε	ε
$\boldsymbol{E'}$	$+\,\boldsymbol{T\,E'}$	$*\,E'$	$n\,E'$	$(\,E'$	$\boldsymbol{\varepsilon}$	$\boldsymbol{\varepsilon}$
$\boldsymbol{T}$	ε	$*\,T$	$\boldsymbol{F\,T'}$	$\boldsymbol{F\,T'}$	ε	ε
$\boldsymbol{T'}$	$\boldsymbol{\varepsilon}$	$*\,\boldsymbol{F\,T'}$	$n\,T'$	$(\,T'$	$\boldsymbol{\varepsilon}$	$\boldsymbol{\varepsilon}$
$\boldsymbol{F}$	ε	ε	$\boldsymbol{n}$	$(\,\boldsymbol{E}\,)$	ε	ε

Рисунок 6. Таблица предсказывающего анализатора для грамматики арифметических выражений, дополненная расширенными правилами восстановления.

На приведенном выше рисунке 6 жирным помечены корректные правила перехода в грамматике, остальные ячейки – ошибочные переходы.

Однако, для корректной работы распознавателя с такой таблицей потребуется модификация исходной $LL(1)$-грамматики, заключающаяся в добавлении фиктивных нетерминалов N_a, отвечающих некоторому терминалу $a \in T$:

$$\forall\, a \in T\colon N_a \to\ a$$

Также введем фиктивный нетерминал $N_\$ \to \varepsilon$ для маркера конца ввода. Определяя такие нетерминалы следует также заменить исходные терминалы в грамматике на фиктивные правила.

Смысл данного приема заключается в необходимости построения полного набора ошибочных ребер в графе конфигураций, речь о котором пойдет в следующих пунктах. Для этого удобно иметь маркеры ошибки в таблице перехода и при этом продолжать разбор в случае попадания в ошибочную транзитную вершину.

Заметим, что введение таких нетерминалов сохраняет $LL(1)$-вид грамматики. Данное утверждение основывается на процессе построения множеств $FIRST$ для обычных нетерминалов: при замене терминалов на фиктивные нетерминалы, данные множества не изменяются, а при их конструировании лишь добавляется еще один транзитный шаг $A \Rightarrow^* N_a v \Rightarrow\ av$.

Множества $FIRST(N_t)$ для всех фиктивных нетерминалов состоят из единственного терминала, их породившего, а значит они не пересекаются.

Аналогично, множества $FOLLOW(u)$ останутся неизменными (за исключением фиктивных терминалов) и в таком случае не будут пересекаться с $FIRST(v)$ для правил вида $X \to u \mid v$, что в итоге позволяет судить о том, что расширенная правилами восстановления грамматика останется $LL(1)$.

Таким образом все ячейки таблицы оказываются полностью заполнены и при построении графа конфигураций из каждой конфигурационной вершины будут исходить ребра по всем возможным терминальным символам.

	+	*	n	(	)	$
$\boldsymbol{E}$	$N_+ E$	$N_* E$	$\boldsymbol{T\ E'}$	$\boldsymbol{T\ E'}$	ε	ε
$\boldsymbol{E'}$	$\boldsymbol{N_+\ T\ E'}$	$N_* E'$	$N_n E'$	$N_(E'$	$\boldsymbol{\varepsilon}$	$\boldsymbol{\varepsilon}$
$\boldsymbol{T}$	ε	$N_* T$	$\boldsymbol{F\ T'}$	$\boldsymbol{F\ T'}$	ε	ε
$\boldsymbol{T'}$	$\boldsymbol{\varepsilon}$	$\boldsymbol{N_* F\ T'}$	$N_n T'$	$N_(T'$	$\boldsymbol{\varepsilon}$	$\boldsymbol{\varepsilon}$
$\boldsymbol{F}$	ε	ε	$\boldsymbol{n}$	$\boldsymbol{N_(\ E\ N_)}$	ε	ε
$\boldsymbol{N_+}$	$\boldsymbol{+}$	*	n	(	)	ε
$\boldsymbol{N_*}$	+	$\boldsymbol{*}$	n	(	)	ε
$\boldsymbol{N_n}$	+	*	$\boldsymbol{n}$	(	)	ε
$\boldsymbol{N_(}$	+	*	n	$\boldsymbol{(}$	)	ε
$\boldsymbol{N_)}$	+	*	n	(	$\boldsymbol{)}$	ε
$\boldsymbol{N_\$}$	+	*	n	(	)	$\boldsymbol{\varepsilon}$

Рисунок 7. Таблица предсказывающего анализатора для расширенной грамматики арифметических выражений.

Теперь настало время обсудить алгоритм построения графа конфигураций. Данный алгоритм на вход принимает сведения о грамматике, а именно:

1. Список всех терминалов грамматики.
2. Описание множеств $FIRST$ для всех обычных нетерминалов.

3. Таблицу предсказывающего анализатора, дополненную правилами восстановления при ошибках и фиктивными нетерминалами.

Список терминалов потребуется во время построения всех возможных путей из конфигурационной вершины по каждому терминалу, а описание множеств $FIRST$ в будущем упростит определение финальных вершин.

Говоря о конфигурационных вершинах, можно разделить их на три группы:

- **Транзитные** – не финальные и не псевдо-финальные вершины.
- **Финальные** – вершины с пустым стеком, либо со стеком из нетерминалов, каждый из которых может раскрываться в пустоту.
- **Псевдо-финальные** – вершины, не являющиеся финальными, из которых не имеется перехода по какому-либо из терминальных символов.

Итак, опишем формально алгоритм построения графа конфигураций:

$\boldsymbol{root} \leftarrow new\ Configuration([A])$
$\boldsymbol{stack} \leftarrow new\ Stack()$

Пока $\boldsymbol{stack}$ не пуст:
    $\boldsymbol{node} \leftarrow \boldsymbol{stack}.pop()$
    Если $node$ – конфигурационная вершина и её стек не пуст:
        Для всех $\boldsymbol{t} \in T \backslash \{\$\}$:
            $\boldsymbol{state}, \boldsymbol{error}, \boldsymbol{hasTerm} \leftarrow getNextState(\boldsymbol{node}.state, \boldsymbol{t})$

            Если $\boldsymbol{hasTerm} = false$: $break$;
            Если $\boldsymbol{state}$ пустое, либо раскрывается в пустоту:
                $\boldsymbol{node}.isFinal = true$

            Если существует конфигурация $\boldsymbol{M}$ со стеком $\boldsymbol{state}$:
                Добавить переход из $\boldsymbol{node}$ в $\boldsymbol{M}$ по символу $\boldsymbol{t}$
            Иначе:
            – Проверить возможность свертки для $\boldsymbol{state}$ и $\boldsymbol{node}$

            Если обобщение или вложение не были обнаружены:
                $\boldsymbol{state}.push(new\ Configuration(\boldsymbol{state}, \boldsymbol{node}))$

Вызов функции $\boldsymbol{getNextState}$ подразумевает проход по таблице предсказывающего анализатора, изменяющий символы на верхушке стека (алгоритм предсказывающего разбора).

Здесь умышленно в целях наглядности не определен процесс проверки на наличие вложения или обобщения, поскольку он требует отдельного рассмотрения.

Для начала определим алгоритм проверки на наличие вложения. Пусть на некотором шаге построения графа конфигураций мы просматриваем переход из конфигурационной вершины M по терминальному символу t, состояние стека S при переходе по данному символу непустое и не раскрывается в пустоту, а также не существует вершин с эквивалентным состоянием.

Стоит заметить, что в данном алгоритме стек заполняется слева-направо, то есть самый левый элемент является «верхушкой» стека. Тогда:

$\boldsymbol{parent} \leftarrow \boldsymbol{M}$

$\boldsymbol{stack} \leftarrow \boldsymbol{S}$

Пока $\boldsymbol{parent} \neq null$:

$\boldsymbol{P} \leftarrow \boldsymbol{parent}.state$

Если $\boldsymbol{parent}$ – конфигурация и $\boldsymbol{P}.length < \boldsymbol{S}.legnth$:

$\boldsymbol{pLen} \leftarrow \boldsymbol{getMaxStackPrefix}(\boldsymbol{P}, \boldsymbol{S})$

Если $\boldsymbol{pLen} > 0$:

Если $\boldsymbol{pLen} = \boldsymbol{P}.length$:

Вложение найдено – $\boldsymbol{return\ parent}$

Иначе: $\boldsymbol{parent} \leftarrow \boldsymbol{parent}.parent$

Говоря менее формально, необходимо осуществить поиск среди предков данной вершины до первого let-узла или, в случае отсутствия таковых, до корня

дерева. Поиск считается успешным, если среди предков текущей вершины была найдена конфигурация, удовлетворяющая определению вложения для стеков.

В случае, если вложение было зафиксировано, создается новая let-вершина, верхним потомком которой является найденная родительская конфигурация, а нижним – конфигурация с не совпавшим остатком стека.

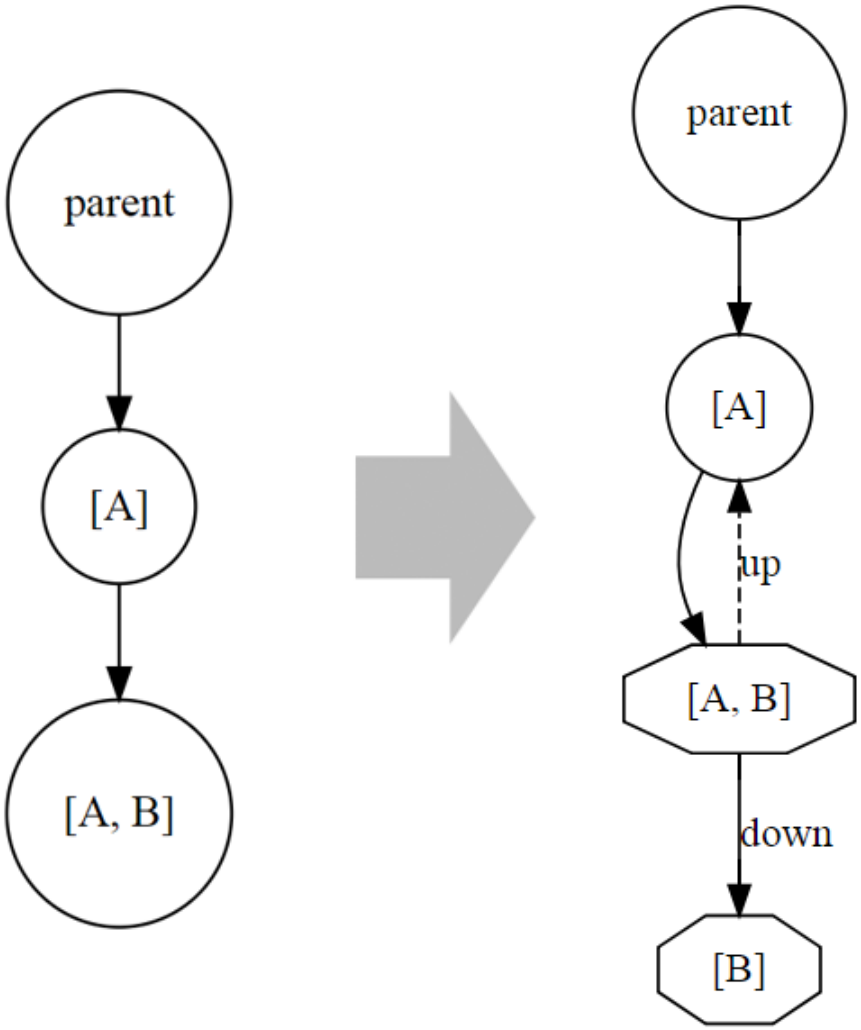

Рисунок 8. Схематичное процесс замены вершины при вложении

В процессе поиска вложения также возможно искать и обобщение. Поскольку эти два механизма являются взаимоисключающими (в случае непустоты подмножеств стека), при наличии общего собственного префикса двух стеков имеет смысл сравнить их суффиксы, и, если максимальные общие суффикс и префикс в итоге образуют родительский стек – обобщение

обнаружено. Приведем вторую часть алгоритма поиска возможности свертки графа в данной вершине:

$\boldsymbol{parent} \leftarrow \boldsymbol{M}$

$\boldsymbol{stack} \leftarrow \boldsymbol{S}$

Пока $\boldsymbol{parent} \neq null$:

$\boldsymbol{P} \leftarrow \boldsymbol{parent}.state$

Если $\boldsymbol{parent}$ – конфигурация и $\boldsymbol{P}.length < \boldsymbol{S}.legnth$:

$\boldsymbol{pLen} \leftarrow \boldsymbol{getMaxStackPrefix}(\boldsymbol{P}, \boldsymbol{S})$

Если $\boldsymbol{pLen} > 0$:

Если $\boldsymbol{pLen} = \boldsymbol{P}.length$:

Вложение найдено – $\boldsymbol{return\ parent}$

Иначе:

$\boldsymbol{sLen} \leftarrow \boldsymbol{getMaxStackSuffix}(\boldsymbol{P}, \boldsymbol{S})$

Если $\boldsymbol{pLen} + \boldsymbol{sLen} = \boldsymbol{P}.length$:

Обобщение найдено – $return\ parent$

Иначе: $\boldsymbol{parent} \leftarrow \boldsymbol{parent}.parent$

Указанные здесь вызовы методов $\boldsymbol{getMaxStackPrefix}$ и $\boldsymbol{getMaxStackSuffix}$ возвращают максимальный общий префикс или суффикс для двух стеков соответственно.

Как и в случае с вложением, при определении обобщения создается новая let-вершина. Однако она подключается не к текущей конфигурации M, а заменяет найденную родительскую вершину. При этом необходимо отбросить построенный ранее подграф.

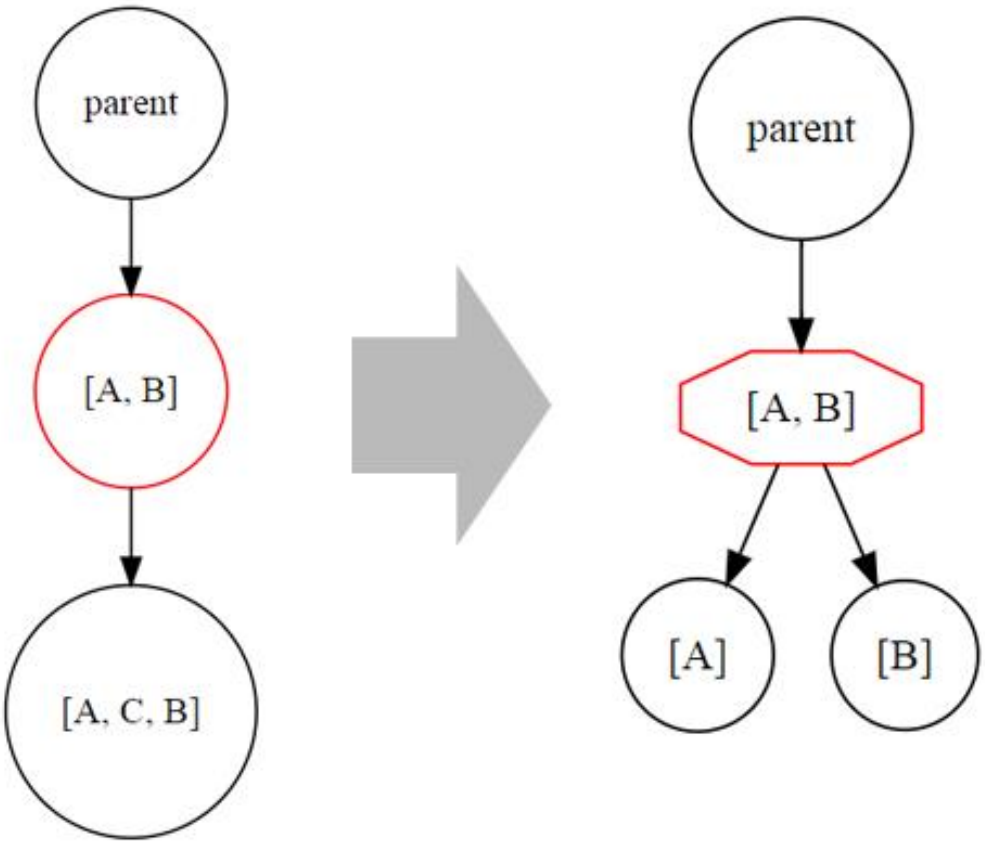

Рисунок 9. Схематичный процесс замены подграфа на let-вершину при обобщении

На рисунке 8 схематично изображен процесс замены подграфа при обнаружении среди состояний родителей подходящего стека. Красным цветом обозначена родительская вершина, в которой сработало обобщение. Таким образом, она заменяется на новую let-вершину с двумя независимыми потомками, а предыдущее поддерево отбрасывается, то есть запускается проход по подграфу и удаление всех посещенных вершин, не образующих циклы.

Стоит заметить, что в процессе построения новых let-узлов необходимо также выделять эквивалентные вершины, а при разделении стеков сначала попытаться отыскать эквивалентные конфигурации, если это возможно.

2.3 ГЕНЕРАЦИЯ ТЕСТОВЫХ ЦЕПОЧЕК

Имея построенный граф конфигураций, возможно обойти его с целью построения некоторого набора цепочек заданной грамматики.

Однако важным вопросом здесь является сам процесс обхода графа конфигураций и критерий окончания это обхода – насколько длинной должна быть выводимая цепочка?

Стоит напомнить, что определенный ранее в разделе «Постановка задачи тестирования» критерий полноты тестирования задавался в контексте таблицы предсказывающего анализатора, что в свою очередь означает посещение в процессе обхода всех ячеек таблицы.

В таком случае данное наблюдение можно переформулировать в терминах графа конфигураций, из каждой конфигурационной вершины которого исходят дуги по всем возможным терминалам:

> *Тестирование синтаксического анализатора (позитивное, негативное) является полным тогда и только тогда, когда в процессе обхода графа конфигураций посещаются все возможные ребра (корректные, ошибочные).*

Под «корректными» ребрами здесь подразумеваются переходы по неошибочным ячейкам таблицы предсказывающего разбора. Ошибочные ребра – ребра, образованные путем перехода в некоторое состояние, в процессе которого была посещена ошибочная ячейка таблицы.

Поскольку результатом работы генератора являются два набора тестов – позитивные и негативные, дальнейшие рассуждения для каждого набора проведем отдельно.

2.3.1 Генерация позитивных тестов

Под позитивными тестами будет понимать набор выводимых из входной грамматики цепочек. Таких цепочек, очевидно, может быть большое количество в зависимости от грамматики, однако основной задачей является построение конечного набора тестов. Имея критерий полноты тестирования для графа конфигураций, можно минимизировать такой набор.

По аналогии с конечными автоматами, как уже было замечено ранее, в графе имеются заключительные (финальные) состояния. В таких состояниях стек раскрывается в пустоту, что означает окончание разбора входной цепочки в контексте детерминированного распознавателя.

Однако в виду специфики данного графа, кроме всего прочего содержащего let-вершины, обойти его стандартными алгоритмами не представляется возможным. Необходимо учитывать переход по let-узлам, и при посещении финальных вершин, в случае отсутствия других вариантов перехода, возвращаться к нижнему потомку и продолжать построение отдельного теста.

При обходе графа с целью построения позитивных тестов выполняются переходы только по нормальным ребрам (не образованным в процессе посещения ошибочной ячейки таблицы разбора).

Начиная обход с корня дерева (графа конфигураций), выполняем поиск ближайшего непомеченного ребра и переходим по нему в следующую вершину. В случае отсутствия непосещенных ребер, если текущая вершина – финальная, построение теста завершается, иначе требуется достроить тест до ближайшей финальной вершины. Это возможно из любого состояния в силу того, что из входной грамматики предварительно удаляются все бесполезные и недостижимые символы. Важно в процессе обхода посетить все ребра и иметь как минимум по одному тесту на каждое финальное состояние (далее данная формулировка будет уточнена для случая с let-вершинами).

Особый интерес здесь представляет поиск ближайшего непомеченного ребра и ближайшей финальной вершины, поэтому в качестве базового алгоритма поиска рассмотрим обход графа в ширину (BFS) [9].

При отсутствии let-вершин процесс обхода в точности совпадает с поиском в ширину: выполняется просмотр переходов из текущей вершины, и, пока не найдено непосещённое ребро и очередь не пуста, добавляем в очередь дочерние конфигурации, соответствующие данным переходам. При этом необходимо также запоминать путь, пройденный до первого непосещенного ребра, чтобы иметь возможность корректно достроить тестовую цепочку до заданной вершины. Аналогично будет происходить и поиск ближайшего финального состояния.

Попробуем теперь обобщить данный алгоритм для возможности обработки let-вершин. Для этого в алгоритме поиска в ширину будем рассматривать не вершины, а стеки. Поскольку переход в let-вершину подразумевает цикл, по окончании которого необходимо перейти по нижней ветке этой let-вершины, при её посещении нижнего и верхнего потомков кладем на стек в заданном порядке. Переход в конфигурационную вершину означает замену вершины на стеке на вершину, соответствующую переходу по ребру.
Таким образом, учитывая вместо вершин посещенные ранее стеки, возможно построить набор позитивных тестов. Алгоритм поиска ближайшей финальной вершины работает аналогично. При попадании в финальную вершину происходит снятие вершины со стека (если возможно) и поиск следующей ближайшей финальной вершины из нижнего потомка let-вершины.

2.3.2 Генерация негативных тестов

По аналогии с генерацией позитивных тестов, генерация негативных также использует поиск в графе в ширину. Однако данная задача более тривиальна, так как не требует остановки в заключительных состояниях и поиска пути до таких состояний.

Заключительными состояниями в данном случае будут уже не финальные вершины, а все конфигурационные, которые не являются финальными, поскольку при генерации негативных тестов стоит задача генерации теста для каждого ошибочного ребра, включая отсутствующие по некоторым терминалам переходы (назовем их «неопределенными» переходами), а также для всех нефинальных вершин. Другими словами, строятся цепочки из дополнения ко входному языку.

Пусть в начале обхода в очереди находится стек, состоящий из единственной корневой вершины графа, а также промежуточный результат генерации некорректной цепочки. Далее, пока очередь не пуста, вынимаем текущий стек и промежуточный результат генерации из очереди. Если стек непуст, то текущей вершиной является узел, снятый с верхушки стека.

Если текущая вершина – конфигурационная, то для каждого ошибочного или неопределенного перехода из неё необходимо добавить новый негативный тест, состоящий из промежуточного результата генерации, к которому приписан терминальный символ по рассматриваемому переходу. Если вершина, кроме всего прочего, не является финальной, то также добавляется негативный тест из промежуточного результата. Для всех остальных переходов требуется добавить в очередь новый стек, получаемый из текущего добавлением вершины, соответствующей переходу и, соответственно, к промежуточным результатам приписать терминальный символ перехода.

Если текущая вершина – let-узел, то, как и в случае с позитивными тестами, в очередь кладется стек, на вершине которого находятся верхний и нижний узлы соответствующей вершины.

Таким образом, путешествуя по графу конфигураций и проходя через все возможные уникальные стеки с ошибочными переходами, возможно построить набор негативных тестов для заданной грамматики.

3. ТЕСТИРОВАНИЕ

При выполнении данной работы был реализован генератор тестов для синтаксического анализатора, получающий на вход грамматику в формате РБНФ и генерирующий для неё позитивный и негативный наборы тестов, а также код представления графа конфигураций на языке DOT.

Процесс тестирования разработанного генератора тестов заключался в подаче ему на вход различных $LL(1)$-грамматик (или приводимых к $LL(1)$ удалением бесполезных символов). На основе таких грамматик строилась таблица предсказывающего анализатора, а также множества FIRST, FOLLOW и список терминальных символов, которые затем подавались на вход построителю графа конфигураций.

Тестирование проходило в несколько этапов:

1. Тестирование построения графа конфигураций
2. Тестировнаие генерации позитивных тестов
3. Тестировнаие генерации негативных тестов

Далее будут рассмотрены механизмы тестирования для каждого этапа.

3.1 ТЕСТИРОВАНИЕ ПОСТРОЕНИЯ ГРАФА КОНФИГУРАЦИЙ

Начальное тестирование построителя графа конфигураций осуществлялось вручную с использованием предварительно построенной таблицей предсказывающего анализа для грамматик, граф которых не содержит let-вершины. В процессе такого тестирования были несколько раз переформулированы алгоритмы построения графа. Также в приложение был добавлен отладочный режим, выводящий в процессе построения графа

технические сообщения, а также генерирующий код графа на языке DOT, что в дальнейшем во много раз упростило процесс тестирования всего приложения.

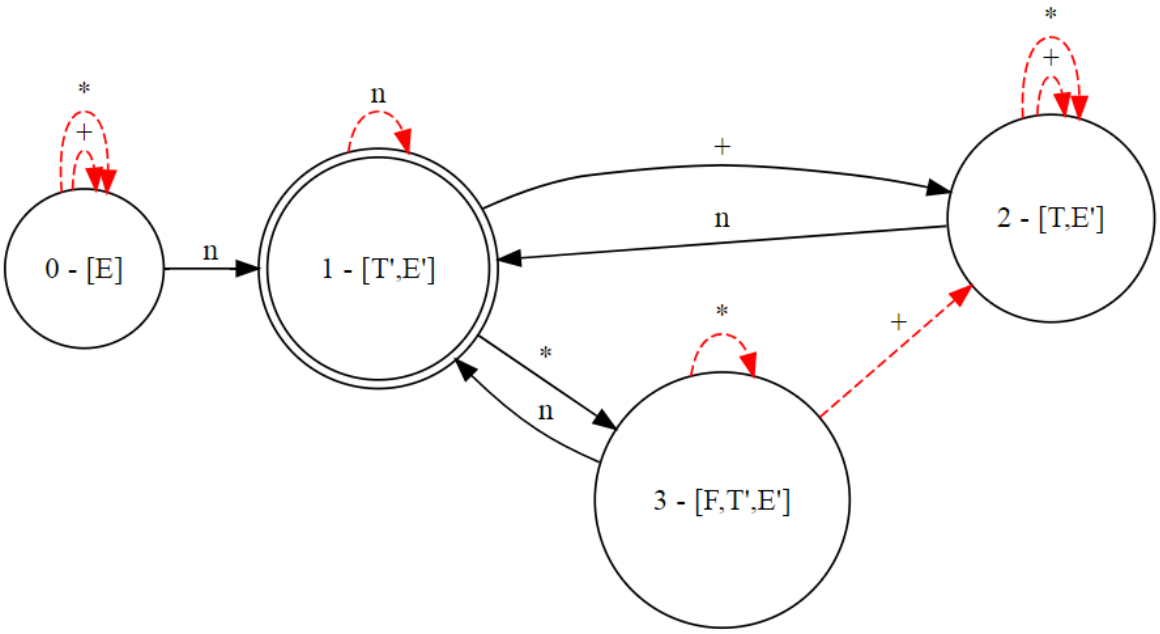

Рисунок 10. Граф конфигураций для грамматики сложений и умножений

На рисунке 10 представлен результат работы построителя графа конфигурации для простой грамматики сложений и умножений. Сплошными стрелками здесь показаны обычные ребра, пунктирными – ребра восстановления.

Далее проводилось тестирование грамматик, в которых граф конфигураций уже содержал let-вершины по вложению. При генерации таких вершин было обнаружено упущение, когда не осуществлялся поиск эквивалентных вершин для дочерних конфигураций let-узла.

На рисунке 11 приведен результат работы построителя графа конфигураций в случае поданной на вход грамматики арифметических выражений. В графе let-вершина изображена многоугольником. Как видно, количество переходов, в том числе ошибочных значительно возрастает с увеличением числа правил грамматики.

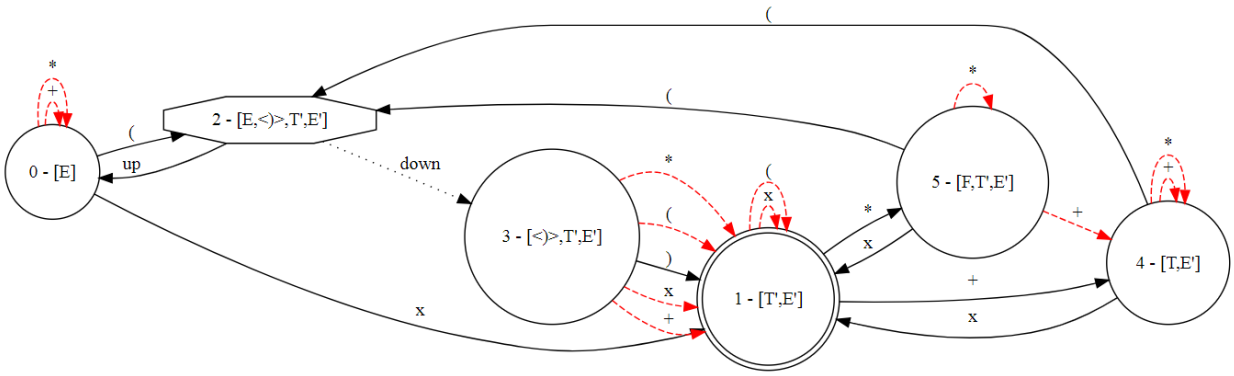

Рисунок 11. Граф конфигураций для грамматики арифметических выражений

Далее было осуществлено тестирование графов с let-вершинами, определяющими вложение. Например, такую грамматику можно получить из грамматики арифметических выражений путем введения новой аксиомы:

$$S \to E\ ';'$$

Такая грамматика называется грамматикой арифметических операторов.

В процессе тестирования ряда грамматик, в которых срабатывает отношение Турчина, были выявлены ошибки, связанные с удалением подграфа и заменой его новой let-вершиной, а также исправлена некорректная функция поиска наличия обобщения.

На рисунке 12 изображен фрагмент графа конфигураций для грамматики арифметических операторов без ребер восстановления для наглядности. Как видно, отношение Турчина встречается практически сразу и расщепляет вычисления на видоизмененный граф арифметических выражений и подграф, содержащий переход по точке с запятой.

Поскольку графы для таких грамматик достаточно объемные, рисунки либо будут опущены, либо приведены частично, возможно без отображения ошибочных ребер.

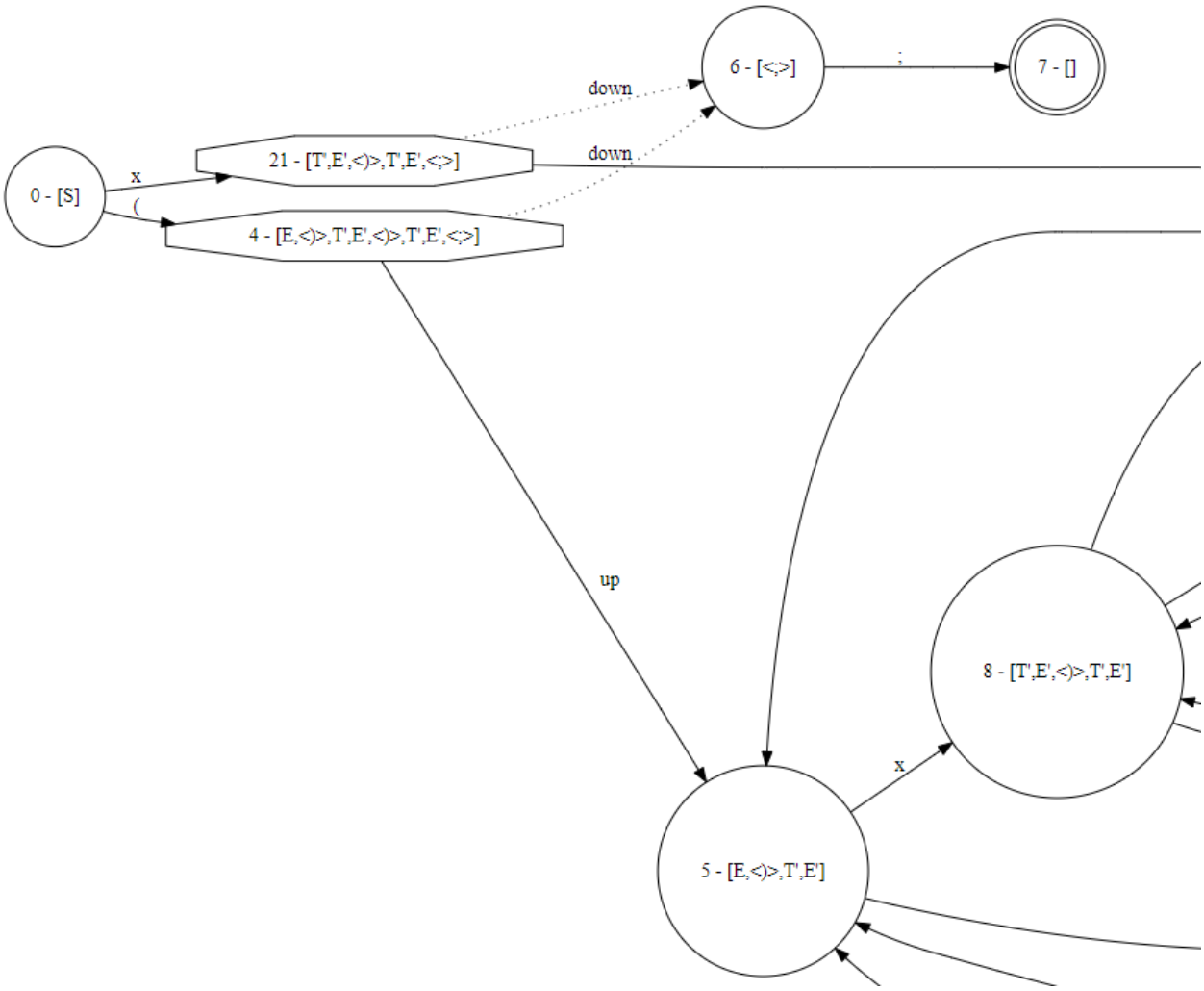

Рисунок 12. Граф конфигураций для грамматики арифметических операторов (без учета ребер восстановления)

Также в процессе тестирования были построены графы конфигураций для языка json, языка определений функций pascal и проведен тест построения графа для собственной грамматики. Были обнаружены некоторые ошибки в алгоритме построения, вызывающие зацикливание.

3.2 ТЕСТИРОВАНИЕ ГЕНЕРАТОРА ПОЗИТИВНЫХ ТЕСТОВ

По окончании тестирования построителя графов была начата разработка алгоритмов обхода полученного графа с целью построения тестов. В процессе разработки сначала рассмотрены позитивные тесты для разного типа грамматик, приведенных в предыдущем пункте.

Генератор позитивных тестов на вход получает корень графа конфигураций, затем обходит его согласно определенному ранее алгоритму построения и в результате генерирует отдельный файл для каждого теста.

Для наглядной оценки корректности построенных тестов в режиме отладки генерировался граф в формате DOT с помеченными жирным пройденными к началу генерации очередного теста вершинами.

Например, для грамматики арифметических выражений граф их нормальных ребер выглядит следующим образом:

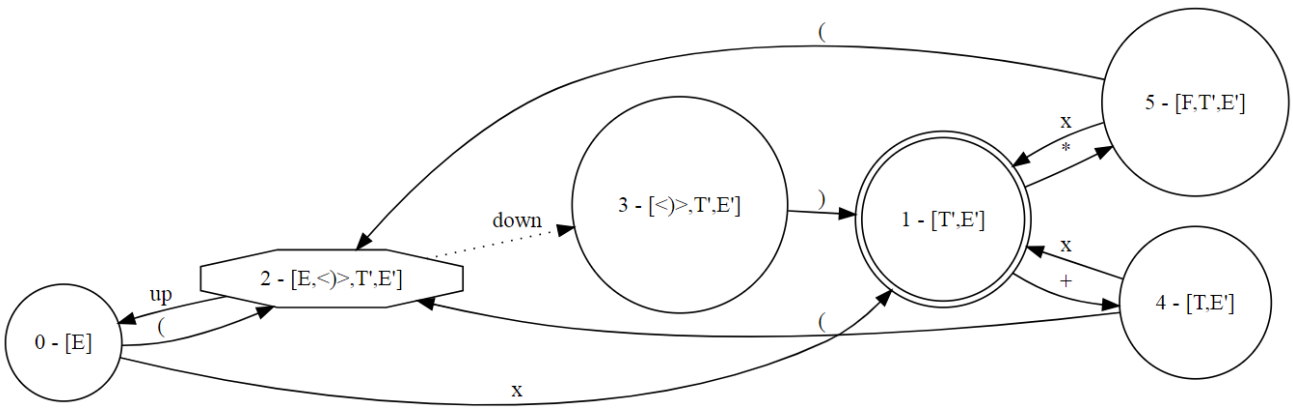

Рисунок 13. Граф конфигураций для грамматики арифметических выражений (без учета ребер восстановления)

В процессе работы генератора были созданы два позитивных теста:

1. $x + x * x$
2. $(x + (x * (x)))$

Действительно, проверим путь стека от начальной вершины до финальной, с учетом критерия остановки:

$$[0] \to {'x'} \to [\mathbf{1}] \to {' + '} \to [4] \to {'x'} \to [\mathbf{1}] \to {' * '} \to [5] \to {'x'} \to [\mathbf{1}] \to \emptyset$$

Стрелки здесь обозначают переход в следующее состояние стеков по очередному символу (либо снятие вершины со стека).

После такого прохода граф конфигураций будет иметь следующий вид:

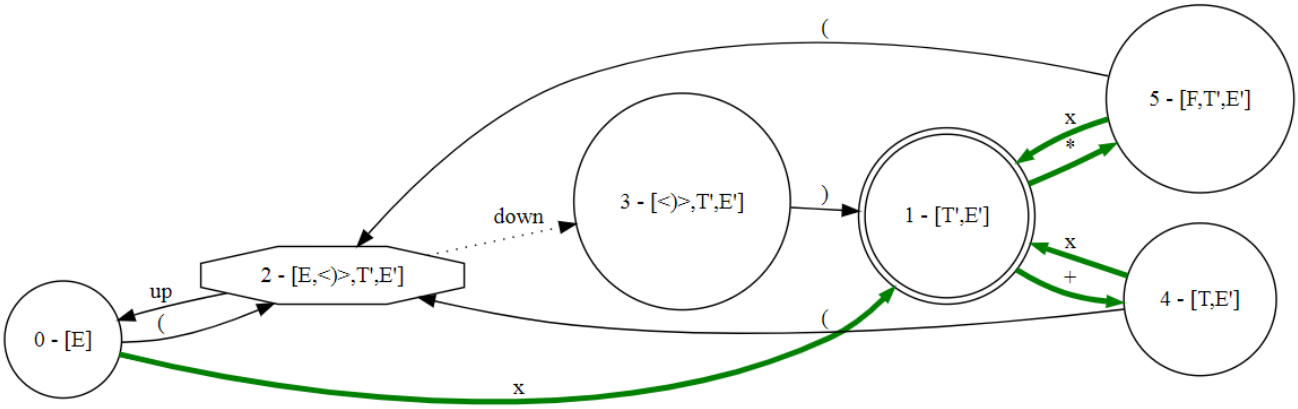

Рисунок 14. Граф конфигураций с помеченными после построения первого позитивного теста ребрами

На рисунке 14 жирным показаны пройденные в процессе построения первого теста ребра. Далее при поиске путей в графе они будут являться транзитными.

Далее повторим проход с изменением стека для следующего теста:

$$[0] \to {'('} \to [0,3] \to {'x'} \to [1,3] \to {' + '} \to [4,3]$$
$$\to {'('} \to [0,3,3] \to {'x'} \to [1,3,3] \to {'*'} \to [5,3,3]$$
$$\to {'('} \to [0,3,3,3] \to {'x'} \to [1,3,3,3] \to [3,3,3] \to {')'} \to [1,3,3]$$
$$\to [3,3] \to {')'} \to [1,3] \to [3] \to {')'} \to [1] \to \emptyset$$

Легко убедиться, что после построения данной цепочки в итоге все корректные ребра графа, а также финальные вершины будут посещены, что означает окончание процесса построения позитивных тестов.

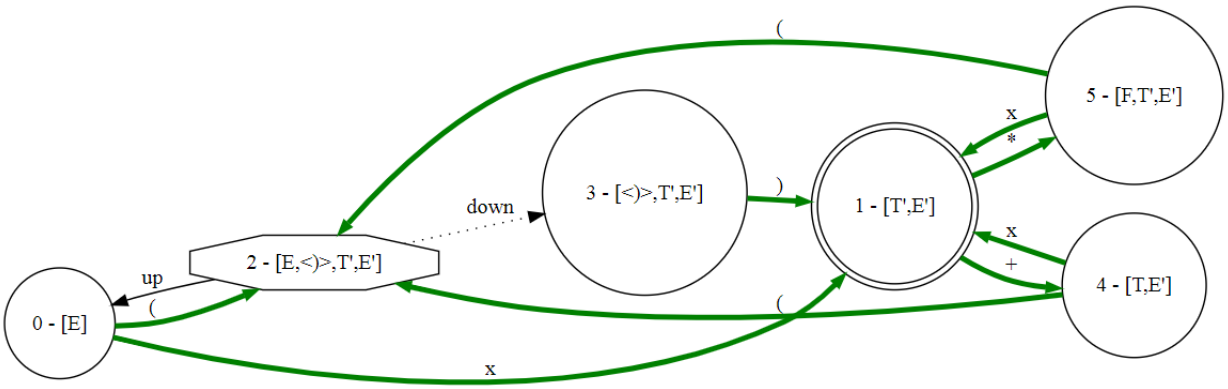

Рисунок 15. Граф конфигураций с помеченными после построения второго позитивного теста ребрами

Тестирование генератора тестов производилось также на следующих грамматиках:

1. **Арифметические операторы**:

   Тест 1. $x + x * x$;

   Тест 2. $(x + x * x)$;

   Тест 3. $(x + (x * (x) + (x) * (x))))$;

2. **JSON**:

   Тест 1. {

   $STRING$: { $STRING$: $STRING$, $STRING$: $NUMBER$},

   $STRING$: [true, false],

   $STRING$: null

   }

   Тест 2. [$STRING$]

3. **Сигнатуры функций Pascal**:

Тест 1. $ident : ident$;

Тест 2. $ident$ ($ident : ident$; $ident$, $ident : char$) $: char$;

Тест 3. $ident : boolean$;

Тест 4. $ident : string$;

Тест 5. $ident : real$;

Тест 6. $ident : integer$;

Тест 7. $ident$ ($ident : boolean$; $ident : string$;

$ident : real$; $ident : integer$): $ident$;

4. **Собственная грамматика**:

Позитивный тест 1.

```
non-terminal A, A;

terminal A, A;

A ::= A * + ? A;
A ::= eps eps ( A A A eps ( eps eps ) * + ? A * + ? ( ( A ( A ) eps ) A )
( A | A A eps ( A * + ? ) ) ) | eps A ;
```

Позитивный тест 2.

```
non-terminal A ;
terminal A ;
A ::= ( A | ( A ) | A A eps ( A ) ) | A eps ( A ) |
      A | eps | ( ( A ) eps ( ( A ) ( ( A ) ) ( ( A ) | A | A | A | eps | ( A ) ) )
) ;
```

Поскольку исходный front-end содержал в себе возможность подачи на вход позитивных и негативных тестов для проверки собственной грамматики, такая проверка была выполнена с использованием сгенерированных тестов. В процессе проверки анализатор ожидаемо

распознал цепочки из набора позитивных тестов и выдал сообщение об ошибке при подаче на вход некорректной цепочки из набора негативных тестов.

Данные примеры лишь показывают возможное применение суперкомпиляции $LL(1)$-грамматик и, очевидно, не могут быть напрямую поданы любому синтаксическому анализатору, в силу того, что в указанных примерах не описываются простейшие конструкции типа $STRING$ или $ident$, то есть приведенные грамматики не достаточно детализированы.

3.3 ТЕСТИРОВАНИЕ ГЕНЕРАТОРА НЕГАТИВНЫХ ТЕСТОВ

Тестирование генератора негативных тестов проходило по схожему с тестированием позитивных тестов сценарию.

В процессе тестирования были рассмотрены уже перечисленные ранее грамматики сложений/умножений, арифметических выражений, операторов, JSON, сигнатур функций языка Pascal, а также собственная входная грамматика.

Поскольку негативных тестов, очевидно, много больше, чем позитивных, далее будут представлены лишь некоторые тесты для достаточно простых грамматик с небольшим количеством переходов.

1. **Грамматика сложений и умножений:**

   Количество негативных тестов: 10

   Тест 1. < пустая строка >

   Тест 2. $+$

   Тест 3. $*$

   Тест 4. $n\ n$

   Тест 5. $n+$

   ……

   Тест 10. $n**$

2. **Грамматика арифметических выражений:**

   Количество негативных тестов: 35

   Тест 1. < пустая строка >

   ……

   Тест 6. n (

   …..

   Тест 19. ( )

   …..

   Тест 35. ($n*$)

3. **JSON:**

   Количество негативных тестов: 187

   Тест 1. < пустая строка >

   ……

   Тест 27. { $STRING\ NUMBER$……

   Тест 174. [[{ $STRING$]

   ……

4. **Собственная грамматика:**

   Количество негативных тестов: 479

   ……

   Тест 21. $non-terminal\ \ non-terminal$

   ……

   Тест 77. $non-terminal\ A;\ \ terminal\ A;$

   ……

   Тест 254. $non-terminal\ A;\ \ terminal\ A; A ::= (\ A\ (\ A\ ;$

   ……

В процессе тестирования данных грамматик были выявлены их недостатки, такие как, например, ошибка разбора примитивных типов в JSON или пустого объекта, а также исправлены недоработки в алгоритме построения негативных тестов.

ЗАКЛЮЧЕНИЕ

В процессе выполнения данной работы были практически изучены основные приемы суперкомпиляции на примере генерации тестов для проверки корректности работы синтаксического анализатора $LL(1)$-грамматик.

Было разработано приложение на языке JavaScript, получающее на вход некоторую грамматику вида $LL(1)$ и строящее для неё необходимые наборы позитивных и негативных тестов.

Полученные результаты в дальнейшем возможно обобщить для более широкого класса грамматик, однако сам по себе полученный результат свидетельствует о том, что, применяя методы суперкомпиляции для решения задачи синтаксического анализа, возможно добиться построения конечного и строго определенного набора тестов, в частности, формализации построения негативных тестов.

Полученные тесты, при корректно сформулированной грамматике, будучи поданными на вход некоторому синтаксическому анализатору, должны быть корректно им обработаны. В противном случае может потребоваться проверка логики работы парсера на предмет ошибок.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. А. Ахо, М. Лам, Р. Сети, Д. Ульман. Компиляторы: принципы, технологии и инструментарий — 2 изд. — М.: Вильямс, 2008.
2. Г. Майерс, Т. Баджетт, К. Сандлер. Искусство тестирования программ — 3 изд. — М.: Вильямс, 2012.
3. K. Hanford. Automatic generation of test cases — IBM Systems Journal, Vol. 9, No. 4, 1970.
4. P. Purdom. A sentence generator for testing parsers — BIT Numerical Mathematics, 1972.
5. С. Зеленов, С. Зеленова. Автоматическая генерация позитивных и негативных тестов для тестирования фазы синтаксического анализа — Институт системного программирования РАН, 2004.
6. А. Непейвода. Отношение Турчина и аппроксимация циклов при анализе программ. Сборник трудов по функциональному языку программирования Рефал. Том №1 — Переславль-Залесский: Сборник, 2014.
7. С. Романенко. Заметки о суперкомпиляции [Электронный ресурс]. Режим доступа: https://sergei-romanenko.github.io/scp-notes-ru/. Дата обращения: 13.03.2018.
8. И.Г. Ключников. Суперкомпиляция: идеи и методы. Практика функционального программирования №7 — Омск, 2011.
9. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ. — 2 изд. — М.: Вильямс, 2005