ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ 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»

ОТЧЕТ ПО ПРЕДДИПЛОМНОЙ ПРАКТИКЕ

«ГЕНЕРАЦИЯ ТЕСТОВ ДЛЯ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА МЕТОДАМИ СИПЕРКОМПИЛЯЦИИ»

Студент группы ИУ9-72 _____/ **Головань С. М.**

Руководитель дипломного проекта _____/ **Коновалов А. В.**

Москва, 2018

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 при их разработке. Ожидается, что в процессе компиляции 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 неотъемлемой частью любого компилятора. Рассматриваемые методы основаны на идее суперкомпиляции, предложенной советским и американским ученым В. Ф. Турчиным в 1986 году в статье «The concept of a supercompiler».

В качестве входного языка часто рассматриваются $LL(1)$-грамматики, преимуществом которых является возможность построения таблицы предсказывающего анализа для выполнения разбора входной цепочки детерминированным распознавателем – автоматом с магазинной памятью.

Целью данной работы является практическое изучение и применение методов суперкомпиляции для построения набора тестовых цепочек парсера $LL(1)$-грамматик.

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.

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 $N \cup T \cup \{\varepsilon\}$.

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 $G = < N, T, P, S >$, а также $x \in T^*$ – 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.

Часто КС-грамматики задаются в расширенной форме Бэкуса-Наура (РБНФ), т.е. имеют вид $X \to R, X \in N, R -$ дерево, задаваемое следующей грамматикой:

$$R ::= \varepsilon \mid X \mid R\ R \mid R\ '|'\ R \mid R'^{*}' \mid R'^{+}' \mid R'^{?}', \text{ где}$$

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

Здесь и далее рассматриваются только контекстно-свободные грамматики в форме РБНФ как наиболее часто используемые и легко формализуемые.

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. Такие грамматики обладают рядом преимуществ, которые будут рассмотрены в данной работе. Но прежде опишем указанный класс более формально.

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

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

$$\boldsymbol{FOLLOW(X)} = \{\alpha \in T \mid S\$ \implies^{*} uX\alpha v\}$$

Таким образом, 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. Указанное 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:

Грамматика G есть $\boldsymbol{LL(1)}$**-грамматика**, 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).

Также, грамматика 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.

Таким образом, исходная грамматика обычно предварительно обрабатывается с целью оптимизации разбора и возможности применения к ним общеизвестных подходов построения синтаксических анализаторов.

1.1.3. Предсказывающий анализ

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

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.

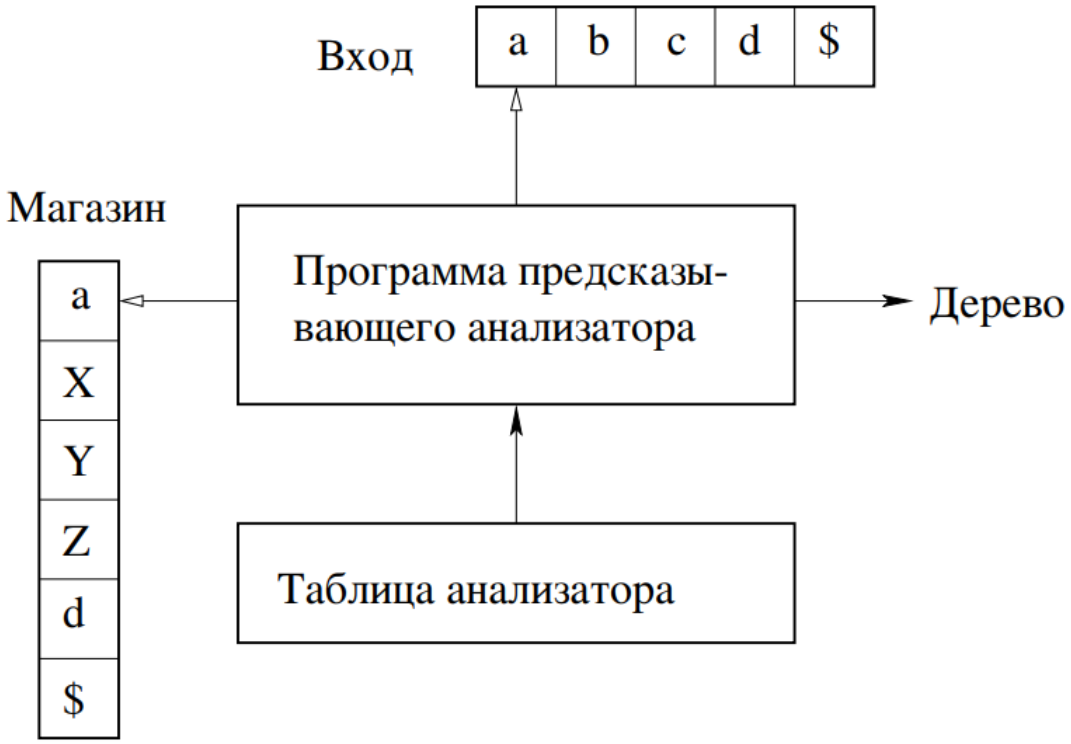

Рисунок 1. Схема работы анализатора

Рассмотрим процесс построения таблицы предсказывающего анализатора. Такая таблица однозначно определяет раскрытие нетерминального правила по текущему символу во входном потоке. В ячейках таблицы располагаются либо правые части правил грамматики, либо индикатор ошибки.

Далее рассматривается детерминированный случай, где в ячейке таблицы может располагаться единственное возможное раскрытие правила по текущему входному символу.

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

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

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ε	'*' F T′	ERROR	ERROR	ε	ε
F	ERROR	ERROR	n	'(' E ')'	ERROR	ERROR

Рисунок 2. Таблица предсказывающего анализатора грамматики арифметических выражений

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.

Рассмотрим восстановление при ошибках «в режиме паники». 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. 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.

Нетер-минал	Входной символ					
	id	+	*	(	)	$
E	$E \to T\ E'$			$E \to T\ E'$	*synch*	*synch*
E'		$E' \to +\ T\ E'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to F\ T'$	*synch*		$T \to F\ T'$	*synch*	*synch*
T'		$T' \to \epsilon$	$T' \to *\ F\ T'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	*synch*	*synch*	$F \to (E)$	*synch*	*synch*

Рисунок 3. Таблица предсказывающего разбора, дополненная правилами восстановления

На Рисунке 3 пустыми ячейками обозначены ошибочные переходы. Значение в ячейке «synch» означает, что в данной ячейке содержится правило восстановления, заданное согласно синхронизирующему множеству для данного нетерминала.

Забегая вперед, 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.

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.

Таким образом все ячейки таблицы полностью заполнены, а в ячейках, содержащих признак ошибки, находится информация, по которой можно продолжить анализ.

1.1.5. Граф состояний анализатора

Поскольку предсказывающий анализатор есть автомат с магазинной памятью, для него можно построить граф процессов (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.

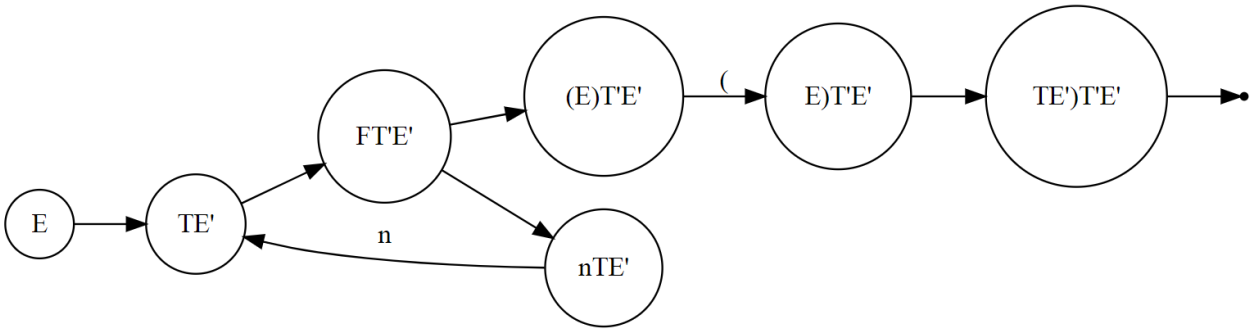

Рисунок 4. Граф состояний грамматики арифметических выражений

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 отсутствие различных вершин с одинаковым состояние магазина существенно упростит процесс свертки.

При использовании режима восстановления при ошибках, из каждой вершины графа состояний будут исходить ребра для каждого терминального символа.

Как станет ясно дальше, именно возможность построения и свертки бесконечного графа, построенного по схожим правилам, а также использование идей суперкомпиляции, позволяют взглянуть на процесс генерации тестовых цепочек для синтаксического анализатора под другим углом.

1.2. Суперкомпиляция

1.2.1. Постановка задачи суперкомпиляции

1.2.2. Дерево конфигураций

1.2.3. Свертка дерева конфигураций

1.2.4. Отношение Турчина

1.2.5. Граф конфигураций

1.3. Постановка задачи тестирования

2. РАЗРАБОТКА

3. ТЕСТИРОВАНИЕ

ЗАКЛЮЧЕНИЕ

СПИСОК ЛИТЕРАТУРЫ