

**Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)**

**ОТЧЕТ**

По лабораторной работе №3

По курсу «Конструирование компиляторов»

Тема: «Синтаксический разбор с использованием метода рекурсивного спуска»

Вариант 5

Студент:

Ковалев М.С.

Группа:

ИУ7-22М

Преподаватель:

Ступников А.А.

Москва 2023

**Теоретическая часть**

Одним из наиболее простых и потому одним из наиболее популярных методов нисходящего синтаксического анализа является метод рекурсивного спуска (recursive descent method).

Метод основан на «зашивании» правил грамматики непосредственно в управляющие конструкции распознавателя. Синтаксические анализаторы, работающие по методу рекурсивного спуска без возврата, могут быть построены для класса грамматик, называющегося LL(1). Первая буква L в названии связана с тем, что входная цепочка читается слева направо, вторая буква L означает, что строится левый вывод входной цепочки, 1 означает, что на каждом шаге для принятия решения используется один символ непрочитанной части входной цепочки. Для строгого определения LL(1) грамматики потребуются две функции - FIRST и FOLLOW. Определение. Пусть α ∈ (Ν ∪ Σ) + , x ∈ Σ + , β ∈ (Ν ∪ Σ) \* . Для КС-грамматики G = (Ν, Σ, P, S) определена функция

FIRSTk(α)={x | α =>\* xβ и |x| = k или α =>\* x и |x| < k } ∪ { ε if α =>\* ε } Иначе говоря, множество FIRSTk(α) состоит из всех терминальных префиксов длины k (или меньше, если из α выводится терминальная цепочка длины, меньшей k) терминальных цепочек, выводимых из α. По определению полагают, что FIRSTk (ε ) = { ε }. Определение. Пусть α, γ ∈ (Ν ∪ Σ) \* , β ∈ (Ν ∪ Σ) + , x ∈ Σ + .

Для КС-грамматики G = (Ν, Σ, P, S) определена функция FOLLOWk(β) = { x | S =>\* αβγ and x ∈ FIRSTk (γ) } ∪ { ε if S =>\* αβ } Иначе говоря, множество FOLLOWk(β) состоит из всех цепочек длины k (или меньше) терминальных цепочек, которые могут встречаться непосредственно справа от β в каких-нибудь цепочках, выводимых из аксиомы, причем если αβ выводимая цепочка, то ε тоже принадлежит FOLLOWk(β). Для грамматики LL(1) k=1, и имеет смысл говорить только о функциях FIRST1(α) и FOLLOW1(β), а вместо фразы «терминальных цепочек» говорить «терминальных символов».

Теорема. КС-грамматика G = (Ν, Σ, P, S) является LL(1)-грамматикой тогда и только тогда, когда для каждых двух различных правил A → β и A → γ выполняется условие FIRST1 (β FOLLOW1 (A)) ∩ FIRST1 (γ FOLLOW1 (A)) = ∅ при всех A ∈ Ν.

**Практическая часть**

В методе рекурсивного спуска полностью сохраняются идеи нисходящего разбора, принятые в LL(1)- грамматиках:

• происходит последовательный просмотр входной строки слева-направо;

• очередной символ входной строки является основанием для выбора одной из правых частей правил группы при замене текущего нетерминала;

• терминальные символы входной строки и правой части правила «взаимно уничтожаются»;

• обнаружение нетерминала в правой части рекурсивно повторяет этот же процесс. В методе рекурсивного спуска эти идеи претерпевают следующие изменения:

• каждому нетерминалу соответствует отдельная процедура (функция), распознающая (выбирающая и «закрывающая») одну из правых частей правила, имеющего в левой части этот нетерминал (т.е. для каждой группы правил пишется свой распознаватель);

• во входной строке имеется указатель (индекс) на текущий «закрываемый символ». Этот символ и является основанием для выбора необходимой правой части правила. Сам выбор «зашит» в распознавателе в виде конструкций if или switch. Правила выбора базируются на построении множеств выбирающих символах, как это принято в LL(1)-грамматике;

• просмотр выбранной части реализован в тексте процедуры-распознавателя путем сравнения ожидаемого символа правой части и текущего символа входной строки;

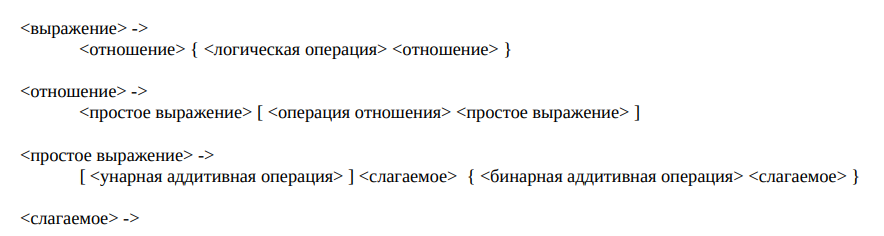
• если в правой части ожидается терминальный символ, и он совпадает с очередным символом входной строки, то символ во входной строке пропускается, а распознаватель переходит к следующему символу правой части;

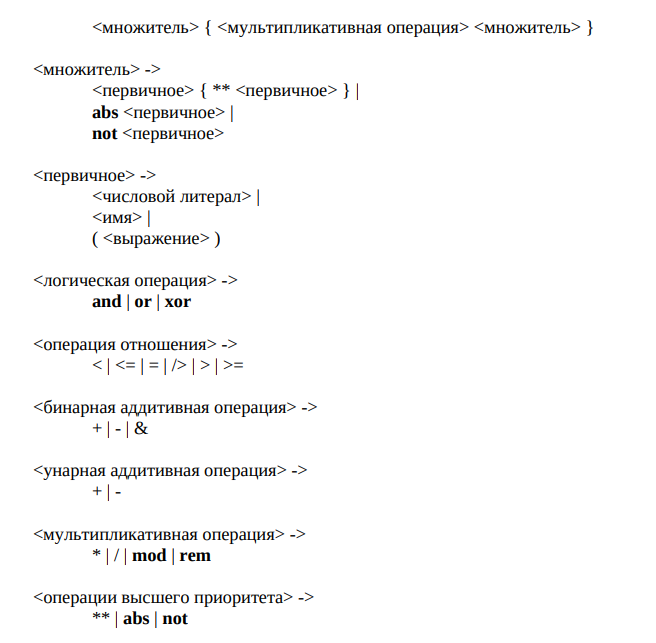
• несовпадение терминального символа правой части и очередного символа входной строки свидетельствует о синтаксической ошибке;

• если в правой части встречается нетерминальный символ, то для него необходимо вызвать аналогичную распознающую процедуру (функцию). Использование рекурсивного спуска позволяет достаточно быстро и наглядно писать программу распознавателя на основе имеющейся грамматики. Главное, чтобы последняя соответствовала требуемому виду. Естественно, возникает вопрос: если грамматика не является LL(1), то существует ли эквивалентная КСграмматика, для которой метод рекурсивного спуска применим? К сожалению, нет алгоритма, отвечающего на поставленный вопрос, т.е. это алгоритмически неразрешимая проблема. Чтобы применить метод рекурсивного спуска, необходимо преобразовать грамматику к виду, в котором множества FIRST не пересекаются. Этот процесс может оказаться сложным.

Поэтому на практике часто используется прием, называемый рекурсивным спуском с возвратами. Для этого лексический анализатор представляется в виде объекта, у которого помимо традиционных методов scan, next и т. п., есть также копирующий конструктор. Затем во всех ситуациях, где может возникнуть неоднозначность, перед началом разбора запоминается текущее состояние лексического анализатора (т.е. заводится копия лексического анализатора) и делается попытка продолжить разбор текста, считая, что рассматривается первая из возможных в данной ситуации конструкций.

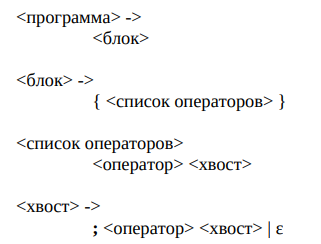
Если этот вариант разбора заканчивается неудачей, то восстанавливается состояние лексического анализатора и делается попытка заново разобрать тот же самый фрагмент с помощью следующего варианта грамматики и т. д. Если все варианты разбора заканчиваются неудачно, то вызывается функция обработки и нейтрализации ошибки. Такой метод разбора потенциально медленнее, чем рекурсивный спуск без возвратов, но в этом случае удается сохранить грамматику в ее оригинальном виде и сэкономить усилия программиста

**Вариант 5.** 



**Задание на лабораторную работу**

Дополнить грамматику блоком, состоящим из последовательности операторов присваивания. Для реализации предлагаются два варианта расширенной грамматики. Вариант в стиле Си



**Текст программы**

<https://github.com/martel42/compilers/tree/master/Lab2>

**Тесты**

Для входной строки '{ c := abs ( - p + 2 & 4 mod 5 / ( 2 ) ) }' программа выводит следующий вывод:

Tokens

['{', 'c', ':=', 'abs', '(', '-', 'p', '+', '2', '&', '4', 'mod', '5', '/', '(', '2', ')', ')', '}']

identifier

un add operation

primary

factor

term

bin add operation

primary

factor

term

bin add operation

primary

factor

mul operation

primary

factor

mul operation

primary

factor

term

simple expression

relation

expression

primary

factor

term

simple expression

relation

expression

primary

factor

term

simple expression

relation

expression

operator

tail

operator\_list

block

program

**Вывод**

Дерево разбора строится достаточно быстро для строки токенов не более 100 символов. Для ускорения процесса создания дерева можно попробовать разбить строку токенов на подстроки и разбирать их от их корня, а не от общего. Данная программа рассчитана на полный перебор всех возможных вариантов построения дерева. В качестве левого правила выбирается сначала нижнее правило из списка, а далее если оно не подошло выбираются следующие. Работа может быть использована для создания ATL дерева.