ВВЕДЕНИЕ

Компилятор – это программа, которая преобразует программный код, написанный на каком-либо языке программирования, в код машинный. Преобразование программного кода в машинный называется компиляцией.

В силу особенностей структуры процессоров, машинный код не универсален, а, следовательно, для компиляции одного и того же кода под разные процессоры требуются разные компиляторы (или универсальный компилятор, способный компилировать код под различные процессоры). То же самое касается и взаимодействия компиляторов с операционными системами: одна и та же программа может работать на одной ОС, но не запустится на других.

На данный момент самыми известными являются такие языки, как C++, Java, C# и Python. Все они используют один и тот же компилятор — GCC (gcc), за исключением языка C++, который, в свою очередь, использует CLAN (Common Language Infrastructure for Objective-C).

Для решения большинства задач встроенных средств уже существующих языков программирования будет достаточно, но решение более специфичных проблем может потребовать разработки собственного языка программирования, и, как следствие, компилятора для него. Кроме того, опыт разработки собственного компилятора позволит на более глубоком уровне понять устройство уже существующих компиляторов. И, наконец, если у разработчиков есть собственное представление о том, какой компилятор им необходим, то процесс создания такого компилятора будет упрощен. Этим определяется актуальность темы иссследования.

Объект исследования – языки программирования.

Предмет исследования – алгоритмы построения компиляторов языков программирования.

Целью данной курсовой работы является изучение различных аспектов функционирования компиляторов и особенностей их разработки.

Из цели исследования вытекают следующие задачи:

1. изучить необходимую литературу по теории создания компилятора;
2. разработать грамматику модельного языка программирования по варианту задания;
3. описать алгоритм лексического анализа и программно реализовать;
4. описать алгоритм синтаксического анализа (и программную реализацию);
5. описать алгоритм семантического анализа (и программную реализацию);
6. разработать алгоритма перевода в ПОЛИЗ;
7. описать алгоритм трансляции в ассемблер.

Работа выполняется по варианту модельного языка №1.

Синтаксис начала программы:

**<программа>** ::= program var <описание> begin <оператор>; {<оператор>;} end.

Синтаксис оператора описания:

**<описание>** ::= <идентификатор> {, <идентификатор>} : <тип>;

Синтаксис идентификатора:

**<идентификатор>** ::= <буква><непустая последовательность цифр><буква>

Синтаксис типа:

**<тип>** ::= int | float | bool

Синтаксис составного оператора:

**<составной оператор>** ::= ”{“ {<оператор>} “}”

Синтаксис оператора присваивания:

**<присваивание>** ::= <идентификатор> = <выражение>

Синтаксис условного оператора:

**<условный оператор>** ::= if <булево выражение> then <оператор> [else <оператор>]

Синтаксис оператора фиксированного цикла:

**<оператор фиксированного цикла>** ::= for <присваивание> to <булево выражение> do <оператор>

Синтаксис оператора условного цикла:

**<оператор условного цикла>** ::= while <булево выражение> do <оператор>

Синтаксис оператора вывода:

**<оператор ввода>** ::= read(<идентификатор>{, <идентификатор>})

Синтаксис оператора вывода:

**<оператор вывода>** ::= write (<выражение>{, <выражение>})

Синтаксис начала и конца комментария:

**<признак начала комментария>** ::=/\*

**<признак конца комментария>** ::=\*/

1. Разработка грамматики модельного языка программирования
   1. Форма Бэкуса-Наура

Форма Бэкуса-Наура (БНФ) представляет собой формальную систему описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие.

БНФ, разработанная на основании варианта задания имеет следующий вид:

**<**программа**>** ::= program [<секция переменных>] <секция кода>

<секция переменных> ::= {var <описание>}

<секция кода> ::= begin <оператор>; {<оператор>;} end

**<**описание**>** ::= <идентификатор> {, <идентификатор>} : <тип>;

**<**идентификатор**>** ::= <буква>[{<цифра> | <буква>}]

**<**тип**>** ::= int | bool

**<**оператор**>** ::= <составной оператор> | <присваивание> | <условный <оператор> | <оператор фиксированного цикла> | <оператор условного цикла> | <оператор ввода> | <оператор вывода>

**<**составнойоператор**>** ::= ”{“ {<оператор>} “}”

**<**присваивание**>** ::= <идентификатор> = <выражение> ;

**<**условныйоператор**>** ::= if <выражение> then <оператор> [else <оператор>]

**<**операторфиксированногоцикла**>** ::= for <присваивание> to <выражение> do <оператор>

<оператор условного цикла> ::= while <булево выражение> do <оператор>

**<**операторввода**>** ::= read(<идентификатор>{, <идентификатор>});

**<**операторвывода**>** ::= write(<выражение>{, <выражение>});

<выражение> ::= <выражение> + <выражение>

| <выражение> - <выражение>

| <выражение> \* <выражение>

| <выражение> / <выражение>

| <выражение> ^ <выражение>

| <выражение> or <выражение>

| <выражение> and <выражение>

| <выражение> > <выражение>

| <выражение> < <выражение>

| <выражение> >= <выражение>

| <выражение> <= <выражение>

| <выражение> != <выражение>

| “(“ <выражение> “)”

| not “(“ <выражение>“)”

| <булева константа>

| <число>

| <идентификатор>

<признак начала комментария> ::= //

**<**булеваконстанта**>** ::= true | false

**<**число**>** ::**=** 0 | [-] <цифра без нуля> {<цифра>}

**<**цифрабезнуля**>** ::**=** 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**<**цифра**>** ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<буква> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

* 1. Формальная грамматика

Формальная грамматика G для данного варианта задания имеет вид:

G = <T, N, S, P>, где T – алфавит терминальных символов, N – алфавит нетерминальных символов, S – начальный символ, P – набор правил вывода.

Число в квадратных скобках рядом с некоторыми нетерминальными символами в множестве P обозначает степень приоритета данного правила. Чем выше это число, тем выше приоритет соответствующего правила относительно остальных.

T = {“program”, “var”, “begin”, “end”, “int”, “bool”, “if”, “then”, “else”, “for”, “to”, “do”, “while”, “read”, “write”, “not”, “or”, “and” “+”, “-”, “\*”, “/”, “^”, “<”, “>”, “==”, “>=”, “<=”, “;”, “{“, “}”, “:”}

N = {

start = <программа>,

var\_section = <секция переменных>

code\_section = <секция кода>

declaration = <описание>,

ID = <идентификатор>,

type = <тип>,

operator = <оператор>,

block\_op = <составной оператор>,

assign\_op = <присваивание>,

if\_op = <условный оператор>,

for\_op = <оператор фиксированного цикла>,

while\_op = <оператор условного цикла>,

read\_op = <оператор ввода>,

write\_op = <оператор вывода>,

expression = <выражение>

}

S = start

P = {

start -> PROGRAM program\_repetition code\_section

var\_section -> VAR var\_section\_repetition\_plus

code\_section -> BEGIN code\_section\_repetition\_plus END

declaration -> id\_list : type ;

type -> INT

type -> BOOL

operator -> assign\_op

operator -> if\_op

operator -> for\_op

operator -> while\_op

operator -> read\_op

operator -> write\_op

operator -> block\_op

assign\_op[@2] -> ID = expression ;

if\_op[@11] -> IF ( expression ) THEN operator

if\_op[@11] -> IF ( expression ) THEN operator ELSE operator

for\_op -> FOR assign\_op TO expression DO operator

while\_op -> WHILE expression DO operator

read\_op[@11] -> READ ( id\_list ) ;

write\_op[@11] -> WRITE ( expression\_list ) ;

block\_op -> { block\_op\_repetition }

expression[@7] -> expression + expression

expression[@7] -> expression - expression

expression[@8] -> expression \* expression

expression[@8] -> expression / expression

expression[@9] -> expression ^ expression

expression[@3] -> expression OR expression

expression[@4] -> expression AND expression

expression[@6] -> expression > expression

expression[@6] -> expression < expression

expression[@5] -> expression == expression

expression[@6] -> expression >= expression

expression[@6] -> expression <= expression

expression[@5] -> expression != expression

expression[@11] -> ( expression )

expression[@10] -> NOT ( expression )

expression -> BOOL\_CONST

expression -> INT\_CONST

expression -> ID

id\_list -> id\_list , ID

id\_list -> ID

expression\_list -> expression\_list , expression

expression\_list -> expression

program\_repetition[~] ->

program\_repetition -> program\_repetition var\_section

var\_section\_repetition\_plus -> declaration

var\_section\_repetition\_plus -> var\_section\_repetition\_plus declaration

code\_section\_repetition\_plus -> operator

code\_section\_repetition\_plus -> code\_section\_repetition\_plus operator

block\_op\_repetition[~] ->

block\_op\_repetition -> block\_op\_repetition operator

}