

Cairo University

Faculty of Engineering

Department of Computer Engineering



**Project Documentation**

**Compiler C**

**Introduction**

In this document I present the approach I made to make a compiler for language similar to c/c++.

Here are the five big modules I had to make:-

* Lexer
* Parser
* Symbol Table
* Quadrable “assembly” language generation
* Semantic analysis

**Compiler**

I implemented in the compiler:-

|  |  |
| --- | --- |
| Variables and Constants declaration | int x; |
| const int x; |
| float y; |
| const float y; |
| Mathematical expressions | x + y |
| x – y |
| x \* y |
| -x |
| x ++ |
| x -- |
| logical expressions | x == y |
| x != y |
| x > y |
| x < y |
| x >= y |
| x <= y |
| Assignment statement | x = y;  x = “mathematical expressions” |
| If then else | if ( “logical expression” ) { “stmts” } |
| else if ( “logical expression” ) { “stmts” } |
| else { “stmts”} |
| while | while ( “logical expression” ) { “stmts” } |
| Do while | do { “stmts” } while ( “logical expression” ) |
| For loop | For ( “Assignment statement” ; “logical expression” ; “Assignment statement” ) { “stmts” } |
| Block Structure | Global |
| Function block |
| If block |
| For block |
| Do-while block |
| While block |
| Functions | int add(int n1, int n2){ “stmts”} |
| Function Call | x = add( 1,2); |
| y = add(1.1,2); |
| Function Overloading | int multiply(int n1, int n2){ “stmts”} |
| float multiply(float n1, int n2){ “stmts”} |
| Type Conversion | float [+/\*-] integer => float |
| float/integer [><==] float/integer => bool |

**LEXER**

|  |  |
| --- | --- |
| User code | Lexer output |
| { | { |
| } | } |
| ( | ( |
| ) | ) |
| ; | ; |
| . | . |
| , | , |
| = | = |
| - | MINUS |
| + | PLUS |
| \* | MUL |
| / | DIV |
| <= | LE |
| >= | GE |
| < | LT |
| > | GT |
| == | EQ |
| != | NE |
| ++ | PP |
| -- | MM |
| int | INT |
| float | FLOAT |
| const | CONST |
| if | IF |
| else | ELSE |
| do | DO |
| while | WHILE |
| for | FOR |
| return | RETURN |
| [0-9]+ | INTNUM |
| [0-9]+.[0-9]+ | FLOATNUM |
| [A-Za-z][A-Za-z0-9\_]\* | ID |
| [ \t\c] | - |
| "\n" | - |

**PARSER**

I splited the grammer to three main categories

* Program => Globla Variable Declaration and all the functions and types
* Statements => every statement in the program.
* Expressions => every expression mentioned above + function calls.

|  |  |  |
| --- | --- | --- |
| Program | Program | Declarations Functions |
| Declarations |
| Functions |
| Declarations | Type ID ';' |
| Declarations Type ID ';' |
| Functions | Type ID '(' ')' Stmt\_Group |
| Functions Type ID '(' Parameters ')' Stmt\_Group |
| Type ID '(' Parameters ')' Stmt\_Group |
| Functions Type ID '(' ')' Stmt\_Group |
| Parameters | Type ID |
| Parameters ',' Type ID |
| Args | Expr |
| Args ',' Expr |
| Type | INT |
| FLOAT |
| CONST INT |
| CONST FLOAT |
| Statements | Stmt | ID '=' Expr ';' |
| RETURN ';' |
| RETURN Expr ';' |
| IF '(' Expr ')' Stmt %prec IFX |
| IF '(' Expr ')' Stmt ELSE Stmt |
| FOR '(' ID '=' Expr ';' Expr ';' ID '=' Expr ')' Stmt |
| WHILE '(' Expr ')' Stmt |
| DO Stmt WHILE '(' Expr ')' ';' |
| Stmt\_Group |
| ID PP |
| ID MM |
| ';' |
| Stmt\_Group | '{' Declarations Stmt\_List '}' |
| '{' Declarations '}' |
| '{' Stmt\_List '}' |
| '{' '}' |
| Stmt\_List | Stmt |
| Stmt\_List Stmt |
| Expressions | Expr | Expr MINUS Expr |
| Expr PLUS Expr |
| Expr MUL Expr |
| Expr DIV Expr |
| MINUS Expr %prec UMINUS |
| Expr LE Expr |
| Expr GE Expr |
| Expr GT Expr |
| Expr LT Expr |
| Expr EQ Expr |
| Expr NE Expr |
| '(' Expr ')' |
| ID '(' ')' |
| ID '(' Args ')' |
| INTNUM |
| FLOATNUM |
| ID |

**Quadruple language**

|  |  |
| --- | --- |
| Quadruple | Description |
| ADD X,Y,Z | ADD X + Y AND STORE RESULT IN Z |
| SUB X,Y,Z | SUB X - Y AND STORE RESULT IN Z |
| MUL X,Y,Z | MULTIPLY X \* Y AND STORE RESULT IN Z |
| DIV X,Y,Z | DIVIDE X / Y AND STORE RESULT IN Z |
| MOV X,Y | MOV X TO Y SO Y=X |
| CMPG X,Y,Z | COMPARE IF X GREATER THAN Y SET Z = 1 ELSE SET Z =-1 |
| CMPL X,Y,Z | COMPARE IF X LOWER THAN Y SET Z = 1 ELSE SET Z =-1 |
| CMPGE X,Y,Z | COMPARE IF X GREATER THAN OR EQUAL Y SET Z = 1 ELSE SET Z =-1 |
| CMPLE X,Y,Z | COMPARE IF X LOWER THAN OR EQUAL Y SET Z = 1 ELSE SET Z =-1 |
| CMPE X,Y,Z | COMPARE IF X EQUAL Y SET Z = 1 ELSE SET Z =-1 |
| CMPNE X,Y,Z | COMPARE IF X NOT EQUAL Y SET Z = 1 ELSE SET Z =-1 |
| JIF RES,JUMB\_LABLE | JUMP IF RES > 0 to JUMB\_LABLE |
| JIFN RES,JUMB\_LABLE | JUMP IF RES < 0 to JUMB\_LABLE |
| BIND X , $x | SEND PARAMTER X BY ATTCHING IT TO RESERVED VARIABLES IN MEMORY SPECIALIZED FOR FUNCTIONS |
| CLRQ | CLEAR THE VALUES IN $x CALL IT AFTER FINISHING MOVING PARAMTER TO LOCAL FUNCTION DOMAIN |
| START | START FROM SPECIFIC LABLE MENTIONED ONLY AT THE START OF THE PROGRAM |
| HALT | STOP PROGRAM |
| $x | x COULD BE A VALUE FROM 0 TO N, $0 SPECIAL FOR RETURN POINTER |

**Semantic Analysis**

It will print to you a warning message in the semantic file..

|  |
| --- |
| if rhs is different type from lhs |
| if rhs identifier is not assigned value before it's used |
| If identifier not declared before wether if it's on the rhs or lhs in the same scope |
| tell you how many times the variable declared before in same scope |
| tell you if the function declared before with the name and number of argument and its types and order |
| if compare between bool and int/float |
| if not found a function name matching the calling function - (no function found) |
| if send arguments with different types but same function name and number of argumets - (no function found) |
| if send arguments with different number of arguments - (no function found) |
| if return type of function doesn't match with the identifier - (no function found) |
| Can’t reassign constant number |
| Can’t reassign constant parameter |
| if Identified a variable but not assigend a value later |
| If you have two names of the same type in the same scope |

**Tests**

Test 1

* To print all the “Quadruples language”
* run it by writing in terminal 🡺 make test1

Test 2

* To invoke all the “Compiler” functionality
* run it by writing in terminal 🡺 make test2

Test 3

* To invoke all the “Semantic Analysis”
* run it by writing in terminal 🡺 make test3

Outputs of all these tests are:-

1. symbol\_file
2. assembly\_file
3. semantic\_file