Cairo University

Computer Engineering Department

Faculty of Engineering

Fourth year

LANGUAGES & COMPILERS

Project Document

Team #9 Members

Name	Section	B.N.
Ziad Atef	1	35
Mostafa Elgendy	2	26
Mostafa Wael	2	28
Youssef Atef	2	40

May 2023

Compilers Project

Table of Contents

Introduction

The designed language is a `C` like programming language.

Sample program:

```
const int a = 5;
float b = 6;
print ("Operations:");
if (a == 5) {
    print ("a is 5");
}
else {
    if (b == 6) {
        print ("b is 6");
    }
    else {
        print ("b is not 6");
    }
}
exit;
```

Run Steps

- `yacc -d main.y`: create y.tab.h and y.tab.c
- `lex main.l`: create lex.yy.c

- `gcc -g lex.yy.c y.tab.c -o main`: create main
- `./main`: run main

For convenience, the above steps are combined in a makefile. To run the makefile, type `make <test case name>` in the terminal.

Tools and Technologies

1. Lex: It breaks down the input text into a sequence of tokens, which are then passed on to the parser for further processing.
2. Yacc: It takes a sequence of tokens as input and produces a parse tree or an abstract syntax tree (AST) that represents the structure of the input according to the grammar rules.

Tokens

Token	Regex	Description
DIGIT	[0-9]	Number between 0 and 9.
ALPHABET	[a-zA-Z]	Upper case or lower case English letter.
ALPHANUM	[0-9a-zA-Z]	Digit, upper case letter, or lower case letter.
SPACE	[\r\t]	Single space or tab.
NEW_LINE	\n	New line.
INLINE_COMMENT	\/\/.*	Single line comment.
MULTILINE_COMMENT	\/*.**\/	Multi-line comment.
arithmeticOps	[/+*%-]	Arithmetic operators (+, -, *, /, %).
bitwiseOps	[&^~|]	Bit-wise operators (AND, OR, NOT, XOR).
endOfStatement	[;]	Semi-colon to mark the end of any statement.
punctuators	[()={}:,]	Language punctuators.
TRUE	[tT]rue | 1 | [yY]es	True value.
FALSE	[fF]alse| 0 | [nN]o	False value.

Syntax

Data Types

Tha language supports the following data types:

- Integer
- Float
- Boolean
- String

It supports modifiers like `const` as well.

```
const int a = 10;
int b = 20;
float c = 10.5;
bool d = true;
string e = "Hello World";
```

Operators

The language supports the common operators in C.

```
// Arithmetic operators
a = b + c;
a = b - c;
a = b * c;
a = b / c;
a = b % c;
// Bitwise operators
a = b & c;
a = b | c;
a = b ^ c;
a = ~b;
// Logical operators
a = b && c;
a = b || c;
a = !b;
// Relational operators
a = b == c;
a = b != c;
a = b > c;
a = b >= c;
a = b < c;
a = b <= c;
// Shift operators
a = b << c;
a = b >> c;
```

Conditional Statements

The language supports the if-else, if-elif-else, and switch-case statements.

```
int a = 10;
// if statement
if (a == 10) {
    print("if");
```

```
    print("another if");
}
elif (a == 11) {
    print("elif");
    print("another elif");
}
else {
    print("else");
    print("another else");
    if (a == 10) {
        print("if");
        print("another if");
    }
    else {
        print("else");
        print("another else");
    }
}
if (a == 10) {
    print("if");
    print("another if");
}
elif(a == 11) {
    print("else");
    print("another else");
}
// switch-case statement
switch (a) {
    default:
        print("default");
        break;
}
switch (a) {
    case 1:
        print("1");
        break;

    case 2:
        print("2");
        break;

    case 3:
        print("3");
        break;
}

switch (a) {
    case 1:
        print("1");
        break;

    case 2:
```

```
        print("2");
        break;

    case 3:
        print("3");
        break;

    default:
        print("default");
        break;
}
```

Loops

The language supports the while, for, and repeat-until loops.

```
// while loop
a = 0;
while (a < 20) {
    print(a);
    a = a + 1;
}
print(a);
while (a < 20) {
    if (a == 10) {
        print(a);
    }
    a = a + 1;
}
// for loop
for (a=2 ; a<10; a = a+1 ) {
    print(a);
}
for (a=2 ; a<10; a= a+1 ) {
    print(a);
    b = a;
    while (b < 10) {
        if (b == 5) {
            print("hi");
            print(b);
        }

        b = b + 1;
    }
}
// repeat-until loop
a = 0;
repeat {
    print(a);
```

```
    a = a + 1;
    print(a);
} until (a == 1);
repeat {
    print(a);
    a = a + 1;
    if (a == 1) {
        print(a);
    }
} until (a == 1);
```

Functions

The language supports functions with and without parameters.

```
int y (){
    print("y");
    return 1;
}
int x(int a, int b) {
    print("add");
    return a + b;
}
x(1, 2); // function call
a = y(); // function call and assignment
```

N.B.: you can't define a function inside any scope.

Enumerations

The language supports enumerations.

```
enum Color{
    RED=10,
    GREEN,
    BLUE=12,
    RED
};
{
    Color c1;
    Color c2=RED;
    Color c3=3+5;
}
```

Production Rules

- program → statements | functionDef | statements program | functionDef program
- statements → statement | codeBlock | controlstatement | statements codeBlock | statements statement | statements controlstatement
- codeBlock → { statements } | { }
- controlstatement → ifCondition | whileLoop | forLoop | repeatUntilLoop | switchCaseLoop
- statement → assignment | exp | declaration | EXIT | BREAK | CONTINUE | RETURN | RETURN exp | PRINT (exp) | PRINT (STRING)
- declaration → dataType IDENTIFIER | dataType assignment | dataIdentifier declaration
- assignment → IDENTIFIER = exp | IDENTIFIER = STRING | enumDeclaration | enumDef
- exp → term | functionCall | - term | '~' term | NOT term | exp '+' exp | exp '-' exp | exp '*' exp | exp '/' exp | exp '%' exp | exp '|' exp | exp '&' exp | exp '^' exp | exp SHL exp | exp SHR exp | exp EQ exp | exp NEQ exp | exp GT exp | exp GEQ exp | exp LT exp | exp LEQ exp | exp AND exp | exp OR exp
- term → NUMBER | FLOAT_NUMBER | TRUE_VAL | FALSE_VAL |IDENTIFIER | (exp)
- dataIdentifier → CONST
- dataType → INT_DATA_TYPE | FLOAT_DATA_TYPE | STRING_DATA_TYPE | BOOL_DATA_TYPE | VOID_DATA_TYPE
- ifCondition → IF (exp) codeBlock | IF (exp) codeBlock ELSE codeBlock | IF (exp) codeBlock ELIF (exp) codeBlock | IF (exp) codeBlock ELIF (exp) codeBlock ELSE codeBlock
- whileLoop → WHILE (exp) codeBlock
- forLoop → FOR (assignment ; exp ; assignment) codeBlock
- repeatUntilLoop → REPEAT codeBlock UNTIL (exp) ;
- case → CASE exp : statements | DEFAULT : statements
- caseList → caseList case | case
- switchCaseLoop → SWITCH (exp) { caseList }
- functionArgs → dataType IDENTIFIER | dataType IDENTIFIER , functionArgs
- functionParams → term | term , functionParams

- functionDef → dataType IDENTIFIER (functionArgs) codeBlock | dataType IDENTIFIER '(' ')' codeBlock
- functionCall → IDENTIFIER (functionParams) | IDENTIFIER ( )
- enumDef → ENUM IDENTIFIER { enumBody }
- enumBody → IDENTIFIER | IDENTIFIER = exp | enumBody , IDENTIFIER | enumBody , IDENTIFIER = exp
- enumDeclaration → IDENTIFIER IDENTIFIER | IDENTIFIER IDENTIFIER = exp