COMPILERS PROJECT

Compily Compiler

Team 15

Names	Section	B.N.
Ahmed Salama	1	4
Ahmed Maher	1	7
Moamen Hassan	2	14
Mohamed Talaat	2	17

Contents

Introduction

Compily is a simple programming language compiler similar to **C++** syntax.

In ***Compily*** we used ***Flex*** *(fast lexical analyzer generator)* which is a tool for generating lexical analyzers, and we used ***Bison*** which is a parser generator to generate the syntax analyzer.

Overview

In this section, we are going to give a brief descriptions and examples for the lexical, syntax and semantics allowed by ***Compily***.

Data Types

Compily supports the basic data types such as:

- **int:** is an integer number.
- **char:** is a character value data type.
- **string:** is a stream of characters.
- **float:** is a real numeric value.

Variables Declarations

Compily supports variables and constants declaration. The variables can be left with initialization or can be initialized, but constants must be initialized and can't be left without initialization.

Examples of variables declaration:-

```
int x;
const float y = 1.5;
char c = 'a';
string name = "moamen";
```

Figure 1: variable declarations

Expressions

Compily supports add, subtract, multiplication, division, negative and positive operators.

Unary Operators: positive “+”, negative “-“.

Binary Operators: add “+”, subtract “-“, multiplication “*”, division “/”.

The difference between ‘+’ (add) and ‘+’ (positive) is the number of operands, 2 and 1 respectively and the same thing for negative “-“ and subtract “-“.

Errors Detected

Compily identifies some sort of errors that can help the user for re-code their code in another way to make it run successfully.

The errors detected are as following:

1) Constants declared without initialization.
2) Identifier re-declaration.
3) Undeclared variable access.
4) Type mismatch.
5) Use of uninitialized variable.

Lexical Analyzer

Here, we write regular expressions for the tokens we need in the parser.

The most important regular expressions are as follows:-

Token	**Regular Expression**
Interger	[0-9]+
Float	(([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*))
Identifier	[_a-zA-Z]([_a-zA-Z]|[0-9])*
Char_value	(\'.\')
String_value	(\'(.)*\')
DataTypes	“int”, “float”, “string”, “char”
Const	“const”
Operators	‘=’, ‘+’, ‘-‘, ‘*’, ‘/’
Whitespaces	[\t\r\n]+

Syntax Analyzer

Here, we write the grammar using ***bison*** to build the syntax tree.

We used Bison precedence and associativity features to resolve the precedence and associativity of mathematical problem.

The grammar rules are as following:-

// NOTE: UPPER_CASE -> terminals.
// LOWER_CASE -> non-terminals.

$program \rightarrow \boldsymbol{\varepsilon}$

$program \rightarrow stmt\_list$

$stmt\_list \rightarrow stmt$

$stmt\_list \rightarrow stmt\_list\ stmt$

$stmt \rightarrow$ **';'**

$stmt \rightarrow expression$ **';'**

$stmt \rightarrow var\_decl$ **';'**

$var\_decl \rightarrow type\ \boldsymbol{IDENTIFIER}$

$var\_decl \rightarrow \boldsymbol{CONST}\ type\ \boldsymbol{IDENTIFIER}$

$var\_decl \rightarrow type\ \boldsymbol{IDENTIFIER}$ **'='** $expression$

$var\_decl \rightarrow \boldsymbol{CONST}\ type\ \boldsymbol{IDENTIFIER}$ '=' $expression$

// Binary

expression → *expression* **'+ '** *expression*

expression → *expression* **'- '** *expression*

expression → *expression* **'* '** *expression*

expression → *expression* **'/ '** *expression*

expression → *expression* **'= '** *expression*

// Unary

expression → **'+ '** *expression*

expression → **'- '** *expression*

---

type → ***TYPE_CHAR*** | ***TYPE_INT*** | ***TYPE_FLOAT*** | ***TYPE_***STRING

value → ***CHAR*** |***INTEGER*** | ***FLOAT*** | STRING

Code Generation

Here, we generate the quadruples like we study in the slides.

The quadruples are in form:-

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Semantic Errors Examples

Constants declared without initialization.

```
const int x;
```

```
moamen@DESKTOP-LM4QFA7:/mnt/c/Users/Moamen Hassan/Desktop/GitHub/Compily$ make all
tests/test_case0.cpp:3:11: error: uninitialized const 'x'
const int x;
          ^
```

Identifier re-declaration

```
int x = 5;
int x = 4;
```

```
tests/test_case0.cpp:4:5: error: 'int x' redeclared
int x = 4;
    ^
```

Undeclared variable access.

```
int y = z;
```

```
tests/test_case0.cpp:3:9: error: 'z' was not declared in the program
int y = z;
        ^
```

Type mismatch.

```
int y = 11 + 0.5;
```

```
tests/test_case0.cpp:3:12: error: invalid operands of types 'int' and 'float' to binary operator '+'
int y = 11 + 0.5;
          ^
```

Use of uninitialized variable.

```
int x;
int y = x;
```

```
tests/test_case0.cpp:4:9: error: variable or field 'x' used without being initialized
int y = x;
        ^
moamen@DESKTOP-LM4QFA7:/mnt/c/Users/Moamen Hassan/Desktop/GitHub/Compily$
```

Symbol Table

The symbol table holds the information of identifiers such as type, the variable_name.

Example

```
int x = 5;
int y = 3;
int z = 1;
```

```
moamen@DESKTOP-LM4QFA7:/mnt/c/Users/Moamen Hassan/Desktop/GitHub/Compily$ cat tests/test_case0.sym
+-------------------------------------------------+---------------------+
| type                                            | identifier          |
+-------------------------------------------------+---------------------+
| int                                             | x                   |
+-------------------------------------------------+---------------------+
| int                                             | y                   |
+-------------------------------------------------+---------------------+
| int                                             | z                   |
+-------------------------------------------------+---------------------+
```

Tests

To run the test cases:-

Test Case 1	make test1
Test Case 2	make test2
Test Case 3	make test3

Work Load

Name	Part
Ahmed Maher	Lexical Analyzer
Ahmed Salama	Syntax Analyzer
Moamen Hassan	Semantic Errors
Mohamed Talaat	Quadruples, symbol table data structure and cool error logs.