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

**Compilers project**

**Compily Compiler**

**Team 15**

**Contents**

[**Introduction 2**](#_Toc39060837)

[**Overview 2**](#_Toc39060838)

[**Data Types 2**](#_Toc39060839)

[**Variables Declarations 2**](#_Toc39060840)

**[Expressions 3](#_Toc39060841)**

**[Errors Detected 3](#_Toc39060842)**

[**Lexical Analyzer 4**](#_Toc39060843)

[**Syntax Analyzer 5**](#_Toc39060844)

[**Code Generation 7**](#_Toc39060845)

[**Semantic Errors Examples 7**](#_Toc39060846)

[**Constants declared without initialization. 7**](#_Toc39060847)

[**Identifier re-declaration 8**](#_Toc39060848)

[**Undeclared variable access. 8**](#_Toc39060849)

[**Type mismatch. 8**](#_Toc39060850)

[**Use of uninitialized variable. 8**](#_Toc39060851)

[**Symbol Table 9**](#_Toc39060852)

[**Tests 9**](#_Toc39060853)

[**Work Load 10**](#_Toc39060854)

# 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:-

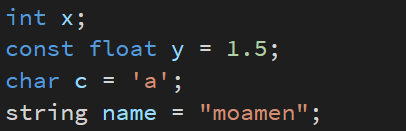


Figure 1: variable declarations

# 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 → **𝜀**

𝑝𝑟𝑜𝑔𝑟𝑎𝑚 → 𝑠𝑡𝑚𝑡\_𝑙𝑖𝑠𝑡

----------------------------------------------------------------------------------------

𝑠𝑡𝑚𝑡\_𝑙𝑖𝑠𝑡 → 𝑠𝑡𝑚𝑡

𝑠𝑡𝑚𝑡\_𝑙𝑖𝑠𝑡 → 𝑠𝑡𝑚𝑡\_𝑙𝑖𝑠𝑡 𝑠𝑡𝑚𝑡

----------------------------------------------------------------------------------------

𝑠𝑡𝑚𝑡 → **'; '**

𝑠𝑡𝑚𝑡 → 𝑒𝑥𝑝𝑟𝑒𝑠𝑠𝑖𝑜𝑛 **'; '**

𝑠𝑡𝑚𝑡 → 𝑣𝑎𝑟\_𝑑𝑒𝑐𝑙 **';'**

----------------------------------------------------------------------------------------

𝑣𝑎𝑟\_𝑑𝑒𝑐𝑙 → 𝑡𝑦𝑝𝑒 **𝐼𝐷𝐸𝑁𝑇𝐼𝐹𝐼𝐸𝑅**

𝑣𝑎𝑟\_𝑑𝑒𝑐𝑙 → **𝐶𝑂𝑁𝑆𝑇** 𝑡𝑦𝑝𝑒 **𝐼𝐷𝐸𝑁𝑇𝐼𝐹𝐼𝐸𝑅**

𝑣𝑎𝑟\_𝑑𝑒𝑐𝑙 → 𝑡𝑦𝑝𝑒 **𝐼𝐷𝐸𝑁𝑇𝐼𝐹𝐼𝐸𝑅** **'=**' 𝑒𝑥𝑝𝑟𝑒𝑠𝑠𝑖𝑜𝑛

𝑣𝑎𝑟\_𝑑𝑒𝑐𝑙 → **𝐶𝑂𝑁𝑆𝑇** 𝑡𝑦𝑝𝑒 **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.

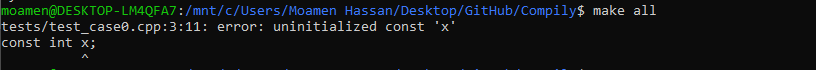
The quadruples are in form:-



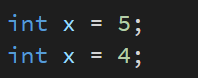
# Semantic Errors Examples

## Constants declared without initialization.





## Identifier re-declaration





## Undeclared variable access.



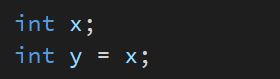


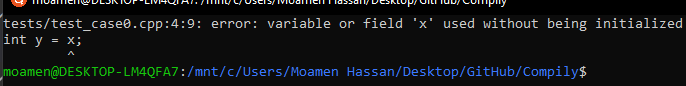
## Type mismatch.





## Use of uninitialized variable.

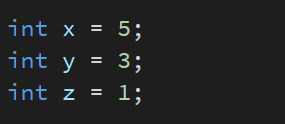


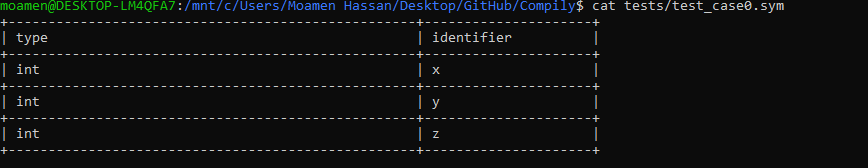


# Symbol Table

The symbol table holds the information of identifiers such as type, the variable\_name.

Example





# 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. |