COSC470 Compiler Design and Implementa

System Manual COSC470\_cyaustria0 Compiler v2

Catherine Austria

for Prof. Mohsen Chitsaz

08/25/2016

# Introduction

This system manual will cover an in depth perspective of the grammar, parser, symbol tables and lexical analyzer and intermediate code generation. Topics to be discussed under grammar are the syntax rules as well as the symbols and operators associated with the grammar. A table will be provided for the grammar. Topics under Parser will include the different types used throughout the program, tables used in the programs like the action and goTo tables, size of the stack and error messages. An in depth look to the symbol tables used will also be provided. Finally, a discussion on the how the compiler works as well as code generation will be provided. An Appendix at the end of the document provides programs to sample using the compiler, their outputs and summaries of error and syntax. Let’s us begin.

# Grammar

## Syntax Rules

The rules for the code are as listed below:

|  |  |  |  |
| --- | --- | --- | --- |
| Rule# |  | Rule# |  |
| 1 | The last executable line of the program must NOT have a ; at the end of its line. | 14 | Character assignments need to be enclosed in ‘ ‘ symbols. |
| 2 | The program must begin with either public or private (access). | 15 | Variable IDs can also be assigned to other variable ID so long as the ID being assigned was declared and initialized. |
| 3 | Variable declarations must come after the method line | 16 | Multiple operators, numbers and IDs can be assigned to a variable ID provided that they are ended with a ; symbol. |
| 4 | Variable declarations must start with var, a space between that and the ID. | 17 | The reserve method in order to get input from a user is get(); |
| 5 | If declaring multiple variables, separate the variables with a comma between them. | 18 | The reserve method in order to display data on the terminal is put(); |
| 6 | The declaring the type for the variable declaration line is denoted by a space after the last ID declared followed by a : sign , a space, the type and ended with a ; | 19 | No operations are allowed between the () of either get or put. |
| 7 | The ; symbol will denote that the line has ended for any lines in the program save for the last line of the program before the } } $ symbols and the special if conditional exception. | 20 | In order to display a variable, the variable must first be initialized via assignment of a int or char. |
| 8 | Variable declarations for a type must all be declared in one line. Multiple instances declaration of the variables throughout the code are not allowed. | 21 | Conditionals take the form of if () lines. |
| 9 | The body pf the code must be in cased in { } symbols. | 22 | Case A Condition: If there is only ONE line to be executed by an if () line, then there is no need to enclose the executed line with { } symbols and the executable line must be placed next to the if () line. |
| 10 | The assignment operator is denoted by the = symbol. | 23 | Case B Condition: If there are more lines to be executed by an if () line, then { } symbols are necessary in order for the program not to crash and ended with a ; so long as the line with the } is not the last line of the program. See Rule 2 for more information. |
| 11 | Operations that can be performed (as well as their respective symbols are as follows: | 24 | Nested ifs are allowed provided that the nested if are only Case A Condition. |
| 12 | Addition (+) ; Subtraction (-) ; Multiplication (\*) ; Division (/) ; Modulo (%) | 25 | The program must end with a $ sign. |
| 13 | Integer assignments to variables can be directly assigned. | 26 | Comments are denoted by // and all characters or symbols after it are ignored. |

Table 1: Syntax Rules

## Symbols and Operators

A general outlook of the grammar of the compiler is shown in the table below. Operators like assignment and comparison are listed near the bottom of the table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Token Code | Used in Grammar Rule | Name of Token | Token Type | Example Code |
| 100 | 1 | start | nonterminal | **access static void ID (identifier\_lis){ declarations compound\_statement } $** |
| 101 | 2,3 | access | nonterminal | |**public**  |**private** |
| 102 | 4,5,6 | identifier\_list | nonterminal | |**ID (specified by user)**  |**Identifier\_list, ID**  |**(nothing)** |
| 103 | 7,8 | declarations | nonterminal | |**declarations var identifier\_list : type ;**  |**(nothing)** |
| 104 | 9,10 | type | nonterminal | |**char**  |**int** |
| 105 | 11 | compound\_statement | nonterminal | **{statement\_list}** |
| 106 | 12,13 | statement\_list | nonterminal | |**statement**  |**statement\_list ; statement** |
| 107 | 14,15,16,17,18 | statement | nonterminal | |**lefthandside**  | **compound\_statement**  |**get(ID)**  |**put(ID)**  |**if(expression) statement** |
| 108 | 19 | lefthandside | nonterminal | **ID = righthandside** |
| 109 | 20 | righthandside | nonterminal | **expression** |
| 110 | 21,22 | expression | nonterminal | |**simple\_ expression**  |**simple\_expression relop simple\_expression** |
| 111 | 23,24 | simple\_expression | nonterminal | |**term**  | **simple\_expression addop term** |
| 112 | 25,26 | term | nonterminal | |**factor**  |**term mulop factor** |
| 113 | 27,28,29,30, 31 | factor | nonterminal | |**ID**  |**num**  |**true**  |**false**  |’**literal**’ |
| 114 | 32,33,34,35,36,  37 | relop | nonterminal | |**>**  |**>=**  |**==**  |**<=**  |**<**  \*|**<>**  \*This is a form of != or not equal to |
| 115 | 38,39 | addop | nonterminal | |**+**  |**-** |
| 116 | 40,41,42 | mulop | nonterminal | |**\***  |**/**  |**%** |
| 1 | 1 | static | Reserve | access **static** void ID(){ |
| 2 | 1 | void | Reserve | access static **void** ID(){ |
| 3 | 1,4,5,16,17,27 | ID | User Defined | |access static void **ID**(){  |**ID**  | Identifier\_list, **ID**  |get(**ID**)  |put(**ID**) |
| 4 | 1,16,17,18 | ( | Reserve | - |
| 5 | 1,16,17,18 | ) | Reserve | - |
| 6 | 1,6 | { | Reserve | - |
| 7 | 1,6 | } | Reserve | - |
| 8 | 1 | $ | Reserve | Signifies end of program |
| 9 | 2 | public | Reserve | public static voidID(){ |
| 10 | 3 | private | Reserve | private static voidID(){ |
| 11 | 5 | , | Reserve | identifier\_list**,** ID |
| 12 | 7 | var | Reserve | **var** ID : int |
| 13 | 7 | : | Reserve | var ID **:** int |
| 14 | 7,13 | ; | Reserve | statement\_list **;** statement |
| 15 | 9 | char | Reserve | var ID : **char** |
| 16 | 10 | int | Reserve | var ID : **int** |
| 17 | 16 | get | Reserve | **get**(ID) |
| 18 | 17 | put | Reserve | **put**(ID) |
| 19 | 18 | if | Reserve | **if** (expression) statement |
| 21 | 28 | num | Reserve | (-∞ to +∞) |
| 22 | 29 | true | Reserve | **true** |
| 23 | 30 | false | Reserve | **false** |
| 35 | 31 | ‘ | Reserve | B = ‘?’; |
| 36 | 31 | literal | Reserve | B = ‘**?**’;  \*This signifies a literal character. Like a = 97, or ? = 63 in Ascii |
| 20 | 19 | = | Assignment | ID **=** 5;  B = ‘?’; |
| 24 | 32 | > | Greater Than | 4>2 |
| 25 | 33 | < | Less Than | 1<VAL1 |
| 26 | 34 | >= | Greater Than or Equal To | 6>=SUM |
| 27 | 35 | <= | Less Than or Equal To | 7<=8 |
| 28 | 36 | == | Equal To | VAL1==VAL2 |
| 29 | 37 | <> | Not Equal To | VAL1<>VAL2 |
| 30 | 38 | + | Addition | 1+VAL2 |
| 31 | 39 | - | Subtraction | 1-VAL2 |
| 32 | 40 | \* | Multiplication | 1\*VAL2 |
| 33 | 41 | / | Division | 1/VAL2  \*Note VAL2 cannot be zero in value |
| 34 | 42 | % | Modulo | 1%VAL2 |

Table 2: Grammar

# Parser

This compiler uses a bottom-up one-pass approach to parsing. The Parser also produces nonterminal tokens, goto tokens and terminal tokens. It also makes use of a hardcoded action table (or parse table) and a goto table. The grammar is a 2D int array that houses all of the grammar from Table 2 into lines that the Parser can pick from based on their indexes. Every grammar line has an int assigned to it that serves as it’s rule number. See the table above for more details on ruling. The size of the stack is dependent on the number of tokens being parsed at the time and error messages that may come up during parsing will also be discussed.

## Types

The compiler produces tokens such as:

* **Nonterminal Tokens** – These are allocated to signify the lefthandside of the grammar rules. These tokens are derived by the compiler using the action table (AKA parse table) and the goto table. These are reduced by the compiler.
* **GoTo Tokens** – These are numbers that are pushed onto the top of the parse stack used to help derive the next action. These are reduced by the compiler. This is hidden in code.
* **ID Tokens** – These tokens are different for they may have a temporary code number that are assigned to them for use in code generation. These temporary code numbers are placements numbers to tell mini and mice where to pass or fetch data from. These can be reduced by the compiler.
* **Terminal tokens** – Most of these are reserve words like “put”, “if” or “get” that are required by the compiler for executing commands. These do not change and can be reduced by the compiler.
* Finally, this compiler only recognizes two kinds of data types for the user program:
* **Int**
* **Char**

## Structure of Tables

### Parse Table

The parse/action table is as seen as below:

private static int[][] *action* = new int[][]{//<editor-fold desc="Too Long for words. Doesn't have Go To">  
/\*0\*/ {8,34,35,4,5,32,30,11,31,33,13,14,25,20,24,6,7,1,2,3,9,10,12,15,16,17,18,19,21,22,23,36,26,28,27,29},  
/\*1\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,1,2,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*2\*/ {-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2},  
/\*3\*/ {-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3},  
/\*4\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*5\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,6,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*6\*/ {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  
/\*7\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,7,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*8\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,8,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*9\*/ {99,99,99,9,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*10\*/ {-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,10,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6}, //er? [9][5]  
/\*11\*/ {-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4},  
/\*12\*/ {99,99,99,99,12,99,99,13,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*13\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,14,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*14\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,15,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*15\*/ {-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8}, //er?[14][15]  
/\*16\*/ {-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5},  
/\*17\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,18,99,99,99,99,99,99,17,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*18\*/ {-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,10,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6},  
/\*19\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,18,99,99,99,21,99,99,99,99,99,22,23,24,99,99,99,99,99,99,99,99},  
/\*20\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,29,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*21\*/ {99,99,99,99,99,99,99,13,99,99,30,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*22\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,31,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*23\*/ {99,99,99,32,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*24\*/ {99,99,99,33,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*25\*/ {99,99,99,34,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*26\*/ {-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15,-15},  
/\*27\*/ {99,99,99,99,99,99,99,99,99,99,99,36,99,99,99,99,35,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*28\*/ {-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12,-12},  
/\*29\*/ {-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14,-14},  
/\*30\*/ {37,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*31\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,38,39,99,99,99,99,99,99,99,99,99,99,99},  
/\*32\*/ {99,99,45,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,41,99,99,99,99,99,99,99,99,42,43,44,99,99,99,99,99},  
/\*33\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,51,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*34\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,52,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*35\*/ {99,99,45,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,41,99,99,99,99,99,99,99,99,42,43,44,99,99,99,99,99},  
/\*36\*/ {-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11,-11},  
/\*37\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,18,99,99,99,21,99,99,99,99,99,22,23,24,99,99,99,99,99,99,99,99},  
/\*38\*/ {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},  
/\*39\*/ {-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9,-9},  
/\*40\*/ {-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10,-10},  
/\*41\*/ {99,99,99,99,99,99,99,99,99,99,99,55,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*42\*/ {-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27,-27},  
/\*43\*/ {-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28,-28},  
/\*44\*/ {-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29,-29},  
/\*45\*/ {-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30,-30},  
/\*46\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,56,99,99,99,99},  
/\*47\*/ {-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19,-19},  
/\*48\*/ {-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20,-20},  
/\*49\*/ {-21,-21,-21,-21,-21,-21,63,-21,64,-21,-21,-21,61,-21,57,-21,-21,-21,-21,-21,-21,-21,-21,-21,-21,-21,-21,-21,-21,-21,-21,-21,58,59,60,62},  
/\*50\*/ {-23,69,-23,-23,-23,67,-23,-23,-23,68,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23,-23},  
/\*51\*/ {-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25,-25},  
/\*52\*/ {99,99,99,99,71,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*53\*/ {99,99,99,99,72,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*54\*/ {99,99,99,99,73,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*55\*/ {-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13,-13},  
/\*56\*/ {-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7},  
/\*57\*/ {99,99,74,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
/\*58\*/ {-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32,-32},  
/\*59\*/ {-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33,-33},  
/\*60\*/ {-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34,-34},  
/\*61\*/ {-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35,-35},  
/\*62\*/ {-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36,-36},  
/\*63\*/ {-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37,-37},  
/\*64\*/ {-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38,-38},  
/\*65\*/ {-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39,-39},  
/\*66\*/ {99,99,45,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,41,99,99,99,99,99,99,99,99,42,43,44,99,99,99,99,99},  
/\*67\*/ {99,99,45,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,41,99,99,99,99,99,99,99,99,42,43,44,99,99,99,99,99},  
/\*68\*/ {-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40,-40},  
/\*69\*/ {-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41,-41},  
/\*70\*/ {-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42,-42},  
/\*71\*/ {99,99,45,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,41,99,99,99,99,99,99,99,99,42,43,44,99,99,99,99,99},  
/\*72\*/ {-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16,-16},  
/\*73\*/ {-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17,-17},  
/\*74\*/ {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,18,99,99,99,21,99,99,99,99,99,22,23,24,99,99,99,99,99,99,99,99},  
/\*75\*/ {-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31,-31},  
/\*76\*/ {-22,-22,-22,-22,-22,-22,63,-22,64,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22,-22},  
/\*77\*/ {-24,69,-24,-24,-24,67,-24,-24,-24,68,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24,-24},  
/\*78\*/ {-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26,-26},  
/\*79\*/ {-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18,-18}  
};

Table 3: Action Table in Parser class

### GoTo Table

The goTo table is as seen as below:

private static int[][] *goTo* = new int[][]{//<editor-fold desc="Too Long for words. has the Go To">  
 {100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116},//0  
 {3,4,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//5  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,11,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//10  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,16,99,99,99,99,99,99,99,99,99,99,99,99,99},//15  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,19,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,20,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,25,26,27,28,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//20  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//25  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//30  
 {99,99,99,99,40,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,46,47,48,49,50,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,53,48,49,50,99,99,99},//35  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,25,99,54,28,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//40  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//45  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,65,66,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,70},//50  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//55  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//60  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//65  
 {99,99,99,99,99,99,99,99,99,99,99,75,49,50,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,76,50,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},//70  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,77,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,25,99,78,28,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,66,99},//75  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,70},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99},  
 {99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,99}//79  
};

Table 4: GoTo table in Parser class

### Grammar

The grammar is as seen as below:

public static int[][] *rules* = new int[][]{//<editor-fold desc="grammar rules">  
/\*0\*/ {0, 0, 0}, //reduced to this  
 { 100, 12, 8, 7, 105, 103, 6, 5, 102, 4, 3, 2, 1, 101},  
 { 101, 1, 9},  
 { 101, 1, 10},  
 { 102, 1, 3 },  
/\*5\*/ { 102, 3, 3, 11, 102},  
 { 102, 0, 0},  
 { 103, 6, 14, 104, 13, 102, 12, 103},  
 { 103, 0, 0},  
 { 104, 1, 15},  
/\*10\*/ { 104, 1, 16},  
 { 105, 3, 7, 106, 6},  
 { 106, 1, 107},  
 { 106, 3, 107, 14, 106},  
 { 107, 1, 108},  
/\*15\*/ { 107, 1, 105},  
 { 107, 4, 5, 3, 4, 17},  
 { 107, 4, 5, 3, 4, 18},  
 { 107, 5, 107, 5, 110, 4, 19},  
 { 108, 3, 109, 20, 3},  
/\*20\*/ { 109, 1, 110},  
 { 110, 1, 111},  
 { 110, 3, 111, 114, 111},  
 { 111, 1, 112},  
 { 111, 3, 112, 115, 111},  
/\*25\*/ { 112, 1, 113},  
 { 112, 3, 113, 116, 112},  
 { 113, 1, 3},  
 { 113, 1, 21},  
 { 113, 1, 22},  
/\*30\*/ { 113, 1, 23},  
 { 113, 3, 35,36,35},  
 { 114, 1, 24},  
 { 114, 1, 26},  
 { 114, 1, 28},  
/\*35\*/ { 114, 1, 27},  
 { 114, 1, 25},  
 { 114, 1, 29},  
 { 115, 1, 30},  
 { 115, 1, 31},  
/\*40\*/ { 116, 1, 32},  
 { 116, 1, 33},  
 { 116, 1, 34}  
};//</editor-fold>

Figure 1: Grammar in Rules class

## Limitation of Stack Size

The limitation of the stack is based on the tokens being parsed at the time. The stack is constantly being pushed and popped from so the size of the stack is dynamic and varies at different points of the program. When the lexical analyzer begins its token making from the file.txt, the parser is yet to be instantiated. After the tokens are made is the parser ready for processing the tokens. Error messages relating to the compiling process is near the end of the document.

# Symbol Tables

## Structure

The symbol table used for the compiler is a Hashtable with a String key and Token element. The Hashtable data structure was used so that there would be quick retrival of the data based on the name of the token being searched. Since the Tokens were mainly text/string based, String was chosen as the key for the table. The symbol table uses Java’s in-built hash function and collision handling. An example of how it is used is shown in the Figure below:

case 4: //{ 102(identifier\_list), 1, 3(ID)}  
 //put ID into symbol table if it isn't already in there  
 Token IDtoken4 = *parseStack*.get(*parseStack*.size() - 2); //get the ID from parseStack  
 if (!IDholder.contains(IDtoken4)) IDholder.add(IDtoken4.tokenName); //for code gen  
 if (!symbolTable.contains(IDtoken4)) symbolTable.put(IDtoken4.tokenName,IDtoken4);  
 //make identifier\_list token and add ID to tokenList of identifier\_list token  
 Token idList;  
 //check if symbol table has identifier\_list token  
 if (symbolTable.contains("identifier\_list")){//if identifier\_list is in the symbol table  
 idList=symbolTable.get("identifier\_list"); //take the "identifier\_list" from the symbol table  
 idList.tokenList.add(IDtoken4); //add ID to that list  
 symbolTable.replace("identifier\_list", idList); //replace the old "identifier\_list" token in the symboltable with the new "identifier\_list" token  
 }else { //if it is not there  
 List<Token> idenlist = new ArrayList<>(); //make a List of Tokens  
 idenlist.add(IDtoken4); //add ID to that list  
 idList=new Token("identifier\_list","","",102, *parseStack*.get(*parseStack*.size() - 2).tokenLine, idenlist); //create the "identifier\_list" token with the list of tokens with ID in it  
 symbolTable.put(idList.tokenName,idList); //put identifier\_list token into symboltable  
 }  
 break;

Figure 2: Use of Symbol Table in Parser for Case 4 in Semantics

# Lexical Analyzer and Intermediate Code Generation

## Assignment Statement

The assignment statement basically checks the types of the ID and righthandside tokens and then performs the assignment to the ID token and updates the lefthandside token as well as the ID in the symboltable. Code Generation on the other hand generates the necessary code associated with the tokens. Code of this process is shown below:

case 19: //{ 108(lefthandside), 3, 109(righthandside), 20(=), 3(ID)} //assignment  
 //If ID is of equal type as righthandside; update ID value and lefthandside  
 String IDname = *parseStack*.get(*parseStack*.size()-6).tokenName; //get the ID key to get from parsestack  
 //if the type of ID is not the same as the type of rhs = error; if same, assign value of rhs to ID  
 Token IDst = symbolTable.get(IDname); //get ID from symboltable  
 Token RHSst = symbolTable.get("righthandside"); //get righthandside from symbol table\*/  
 //code gen   
 if (RHSst.tokenCode == 3) { //if RHSst is an ID op1 is the index of the RHSst name stored in IDholder (direct addressing) and op3 is the index of the IDst name stored in IDholder  
 quadPrint("STO ",String.*valueOf*(IDholder.indexOf(RHSst.tokenName)),"",String.*valueOf*(IDholder.indexOf(IDst.tokenName)));  
 }else if (RHSst.tokenTempCode!=-1){ //if RHSst has a temorary code which was assigned by the codeGenCounter op1 is the temporary code assigned by CodeGenCounter and op3 is the index of the IDst name stored in IDholder  
 quadPrint("STO ",String.*valueOf*(RHSst.tokenTempCode),"",String.*valueOf*(IDholder.indexOf(IDst.tokenName)));  
 }else quadPrint("STO ","#"+RHSst.tokenValue,"",String.*valueOf*(IDholder.indexOf(IDname))); //ALL ELSE op1 immediate data (char or int thus the #) is and op3 is the index of the IDst name stored in IDholder  
  
 //update data in symbol information in symboltable  
 if (IDst!=null && RHSst.tokenType.equals(IDst.tokenType)){  
 symbolTable.get(IDname).tokenValue=RHSst.tokenValue; //update value of ID in symboltable by assigning rhs.value to ID.value  
 Token lhs = new Token("lefthandside",IDst.tokenType,IDst.tokenValue,108, IDst.tokenLine, null); //create lhs token  
 symbolTable.put("lefthandside",lhs); //put token in symboltable  
 System.*out*.println("\n\n\nLHS TYPE PLACED"+"\nID:"+IDst+"\nRHS:"+RHSst+"\n"+symbolTable.get("lefthandside"));//test  
 return lhs.tokenValue;  
 //symTabPrint();//test  
 //return true; //passes semantics  
 }else {  
 if (IDst!=null) Error.*IDTypeMismatch*(IDst.tokenLine, IDst.tokenName, RHSst.tokenType, IDst.tokenType); //error type mismatch  
 //return false; //fails semantics  
 }  
 break;

Figure 3: Assignment Case 19 in Parser class depicting semantic check and code generation.

### Character and Integer Type Assignment

case 9://{ 104(type), 1, 15(char)}  
 //make a token for type and set name to type, type and value to char, code to 104 (nonterminal code for type), linenum to the linenum of the char in the stack and null of the identifier array  
 Token typeTokenChar = new Token("type","char","char",104, *parseStack*.get(*parseStack*.size() - 2).tokenLine, null);  
 //put in symbol table if not already there  
 if (symbolTable.contains("type")) symbolTable.replace("type", typeTokenChar);  
 else symbolTable.put("type",typeTokenChar);   
 break;

case 10://{ 104(type), 1, 16(int)}  
 //make a token for type and set name to type, type and value to int, code to 104 (nonterminal code for type), linenum to the linenum of the int in the stack and null of the identifier array  
 Token typeTokenInt = new Token("type","int","int",104, *parseStack*.get(*parseStack*.size() - 2).tokenLine, null);  
 //put in symbol table if not already there  
 if (symbolTable.contains("type")) symbolTable.replace("type", typeTokenInt);  
 else symbolTable.put("type",typeTokenInt);   
 break;

Figure 4: Character and Integer Type Assignment

## Evaluation of Expressions

The code shown below is categorized via the comparison statements, the addition-subtraction statements and the multiplication-division-modulo statements:

### Comparison Statements

case 22: //{ 110(expression), 3, 111(\*simple\_expression), 114(\*relop), 111(\*simple\_expression)} //ASK ABOUT THIS  
 Token re1=*parseStack*.get(*parseStack*.size()-2); //the first simple\_expression  
 Token re2=*parseStack*.get(*parseStack*.size()-6); //the second simple\_expression  
 Token retok = symbolTable.get("relop"); //fetch operator token off the stack  
 Token termRe; //temporary token for "term" lhs to put in symbol table  
 //semantically test if term and factor are of same type  
 try{  
 Integer firstre = Integer.*parseInt*(re1.tokenValue.trim());  
 Integer secondre = Integer.*parseInt*(re2.tokenValue.trim());  
 if (!re1.tokenType.equals(re2.tokenType)) {  
 System.*out*.println("ERROR: Type error.");  
 System.*exit*(0); //terminate program  
 }  
 String reop1="",reop2=""; //op1 and op2   
 //if the value of either ad1 or ad2 is a number (imediate data) then put # sign infront for op1 and op2  
 if (!IDholder.contains(re1.tokenName)) reop1+="#"+re1.tokenValue; //if ad1 is ! an ID use value for op1  
 if (!IDholder.contains(re2.tokenName)) reop2+="#"+re2.tokenValue; //if ad1 is ! an ID use value for op1  
 //else put the index of the ID from IDholder  
 if (IDholder.contains(re1.tokenName)) reop1+=IDholder.indexOf(re1.tokenName); //if ad1 is an ID use index for op1  
 if (IDholder.contains(re2.tokenName)) reop2+=IDholder.indexOf(re2.tokenName); //if ad2 is an ID use index for op2  
  
 //at this point types are fine take the relop operator and choose action  
 String re=retok.tokenValue;  
 int curLine=*printlist*.size(); //current line in the code gen file  
 int condjump=curLine+3; //op3 for cond instruction quad  
 int jmpjmp=curLine+4; //op3 for the jump after cond instruction quad  
 switch (re){  
 case ">":  
 if (firstre>secondre) termRe = new Token("expression","int",re1.tokenValue, 110, retok.tokenLine,null); //if re1 is bigger than re2; assign value of re1  
 else termRe = new Token("expression","int",re2.tokenValue, 110, retok.tokenLine,null); // assign value of re2  
 termRe.tokenTempCode=codeGenCounter; //assign counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("expression",termRe);  
 //code gen  
 quadPrint("JGT ",reop1,reop2,"#"+String.*valueOf*(condjump));  
 quadPrint("STO ","#1","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JMP ","","","#"+String.*valueOf*(jmpjmp));  
 quadPrint("STO ","#0","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JNE ",String.*valueOf*(termRe.tokenTempCode),"#1",""); //you set op1 to temp code, op2 to #true and op3 to the number from backpatch  
 backStack.push(*printlist*.size()-1); //add the current code gen line to backpatch stack  
 break;  
 case ">=":  
 if (firstre>=secondre) termRe = new Token("expression","int",re1.tokenValue, 110, retok.tokenLine,null); //if re1 is bigger than re2; assign value of re1  
 else termRe = new Token("expression","int",re2.tokenValue, 110, retok.tokenLine,null); // assign value of re2  
 termRe.tokenTempCode=codeGenCounter; //assign counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("expression",termRe);  
 //code gen  
 quadPrint("JGE ",reop1,reop2,"#"+String.*valueOf*(condjump));  
 quadPrint("STO ","#1","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JMP ","","","#"+String.*valueOf*(jmpjmp));  
 quadPrint("STO ","#0","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JNE ",String.*valueOf*(termRe.tokenTempCode),"#1",""); //you set op1 to temp code, op2 to #true and leave op3 to the number from backpatch  
 backStack.push(*printlist*.size()-1); //add the current code gen line to backpatch stack  
 break;  
 case "==":  
 if (firstre==secondre) termRe = new Token("expression","int",re1.tokenValue, 110, retok.tokenLine,null); //if re1 is bigger than re2; assign value of re1  
 else termRe = new Token("expression","int",re2.tokenValue, 110, retok.tokenLine,null); // assign value of re2  
 termRe.tokenTempCode=codeGenCounter; //assign counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("expression",termRe);  
 //code gen  
 quadPrint("JEQ ",reop1,reop2,"#"+String.*valueOf*(condjump));  
 quadPrint("STO ","#1","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JMP ","","","#"+String.*valueOf*(jmpjmp));  
 quadPrint("STO ","#0","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JNE ",String.*valueOf*(termRe.tokenTempCode),"#1",""); //you set op1 to temp code, op2 to #true and leave op3 to the number from backpatch  
 backStack.push(*printlist*.size()-1); //add the current code gen line to backpatch stack  
 break;  
 case "<=":  
 if (firstre<=secondre) termRe = new Token("expression","int",re1.tokenValue, 110, retok.tokenLine,null); //if re1 is bigger than re2; assign value of re1  
 else termRe = new Token("expression","int",re2.tokenValue, 110, retok.tokenLine,null); // assign value of re2  
 termRe.tokenTempCode=codeGenCounter; //assign counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("expression",termRe);  
 //code gen  
 quadPrint("JLE ",reop1,reop2,"#"+String.*valueOf*(condjump));  
 quadPrint("STO ","#1","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JMP ","","","#"+String.*valueOf*(jmpjmp));  
 quadPrint("STO ","#0","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JNE ",String.*valueOf*(termRe.tokenTempCode),"#1",""); //you set op1 to temp code, op2 to #true and op3 to the number from backpatch  
 backStack.push(*printlist*.size()-1); //add the current code gen line to backpatch stack  
 break;  
 case "<":  
 if (firstre<secondre) termRe = new Token("expression","int",re1.tokenValue, 110, retok.tokenLine,null); //if re1 is bigger than re2; assign value of re1  
 else termRe = new Token("expression","int",re2.tokenValue, 110, retok.tokenLine,null); // assign value of re2  
 termRe.tokenTempCode=codeGenCounter; //assign counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("expression",termRe);  
 //code gen  
 quadPrint("JLT ",reop1,reop2,"#"+String.*valueOf*(condjump));  
 quadPrint("STO ","#1","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JMP ","","","#"+String.*valueOf*(jmpjmp));  
 quadPrint("STO ","#0","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JNE ",String.*valueOf*(termRe.tokenTempCode),"#1",""); //you set op1 to temp code, op2 to #true and op3 to the number from backpatch  
 backStack.push(*printlist*.size()-1); //add the current code gen line to backpatch stack  
 break;  
 case "<>": // version of !=  
 if (firstre!=secondre) termRe = new Token("expression","int",re1.tokenValue, 110, retok.tokenLine,null); //if re1 is bigger than re2; assign value of re1  
 else termRe = new Token("expression","int",re2.tokenValue, 110, retok.tokenLine,null); // assign value of re2  
 termRe.tokenTempCode=codeGenCounter; //assign counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("expression",termRe);  
 //code gen  
 quadPrint("JNE ",reop1,reop2,"#"+String.*valueOf*(condjump));  
 quadPrint("STO ","#1","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JMP ","","","#"+String.*valueOf*(jmpjmp));  
 quadPrint("STO ","#0","",String.*valueOf*(termRe.tokenTempCode));  
 quadPrint("JNE ",String.*valueOf*(termRe.tokenTempCode),"#1",""); //you set op1 to temp code, op2 to #true and leave op3 to the number from backpatch  
 backStack.push(*printlist*.size()-1); //add the current code gen line to backpatch stack  
 break;  
 default:  
 System.*out*.println("ERROR: Invalid relop operator parsed at line: "+retok.tokenLine); //error test  
 }  
 }  
 catch (NumberFormatException e){  
 System.*out*.println("Illegal Input. At line: "+retok.tokenLine); //error  
 }  
 break;

Figure 5: Comparison Statements in Parser class

### Addition-Subtraction Statements

case 24: //{ 111(simple\_expression), 3, 112(term), 115(addop), 111(simple\_expression)}  
 Token ad1=symbolTable.get("simple\_expression"); //term  
 Token ad2=symbolTable.get("term"); //simple\_expression  
 Token addtok = symbolTable.get("addop"); //fetch operator token off the stack  
 Token termAd; //temporary token for "term" lhs to put in symbol table   
 //semantically test if term and factor are of same type  
 try{  
 int firstad = Integer.*valueOf*(ad1.tokenValue.trim());  
 int secondad = Integer.*valueOf*(ad2.tokenValue.trim());  
 int resultad;  
 if (!ad1.tokenType.equals(ad2.tokenType)) {//if types are not the same  
 System.*out*.println("ERROR: Type error at line: "+addtok.tokenLine);  
 System.*exit*(0); //terminate  
 }  
 String addop1="",addop2=""; //op1 and op2  
  
 //if the value of either ad1 or ad2 is a number (imediate data) then put # sign infront for op1 and op2  
 if (!IDholder.contains(ad1.tokenName)) addop1+="#"+ad1.tokenValue; //if ad1 is ! an ID use value for op1  
 if (!IDholder.contains(ad2.tokenName)) addop2+="#"+ad2.tokenValue; //if ad1 is ! an ID use value for op1  
 //else put the index of the ID from IDholder  
 if (IDholder.contains(ad1.tokenName)) addop1+=IDholder.indexOf(ad1.tokenName); //if ad1 is an ID use index for op1  
 if (IDholder.contains(ad2.tokenName)) addop2+=IDholder.indexOf(ad2.tokenName); //if ad2 is an ID use index for op2  
 //at this point types are fine take the addop operator and choose action  
 String add=addtok.tokenValue;  
 switch (add){  
 case "+":  
 resultad = firstad+secondad;  
 termAd = new Token(Integer.*toString*(resultad),"int",Integer.*toString*(resultad), 111, addtok.tokenLine,null);  
 termAd.tokenTempCode=codeGenCounter; //assin counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("simple\_expression",termAd); //put in symboltable  
  
 //code gen  
 quadPrint("ADD ",addop1,addop2,String.*valueOf*(termAd.tokenTempCode));  
 return termAd.tokenValue;  
 case "-":  
 resultad = firstad-secondad;  
 termAd = new Token(Integer.*toString*(resultad),"int",Integer.*toString*(resultad), 111, addtok.tokenLine,null);  
 termAd.tokenTempCode=codeGenCounter; //assin counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("simple\_expression",termAd);  
 //code gen  
 quadPrint("SUB ",addop1,addop2,String.*valueOf*(termAd.tokenTempCode));  
 return termAd.tokenValue;  
 default:   
 break;  
 }  
 }catch (NumberFormatException e){  
 System.*out*.println("Illegal Input. At line: "+addtok.tokenLine); //error  
 e.printStackTrace();  
 }catch (NullPointerException e){  
 System.*out*.println("Null Operation. At line: "+addtok.tokenLine); //error  
 e.printStackTrace();  
 }   
 break;

Figure 6: Addition-Subtraction Cases in Parser class

### Multiplication-Division-Modulo Statements

case 26: //{ 112(term), 3, 113(factor), 116(mulop), 112(term)}  
 Token mu1=symbolTable.get("term"); //term  
 Token mu2=symbolTable.get("factor"); //factor  
 Token multok = symbolTable.get("mulop"); //fetch operator token off the stack  
 Token termMu; //temporary token for "term" lhs to put in symbol table  
 try{  
 int firstmu = Integer.*parseInt*(mu1.tokenValue.trim());  
 int secondmu = Integer.*parseInt*(mu2.tokenValue.trim());  
 int resultmu;  
 //semantically test if term and factor are of same type  
 if (!mu1.tokenType.equals(mu2.tokenType)) {  
 System.*out*.println("ERROR: Type error at line: "+multok.tokenLine);  
 System.*exit*(0); //terminate program  
 }  
 String muop1="",muop2=""; //op1 and op2  
 //if the value of either ad1 or ad2 is a number (imediate data) then put # sign infront for op1 and op2  
 if (!IDholder.contains(mu1.tokenName)) muop1+="#"+mu1.tokenValue; //if ad1 is ! an ID use value for op1  
 if (!IDholder.contains(mu2.tokenName)) muop2+="#"+mu2.tokenValue; //if ad1 is ! an ID use value for op1  
 //else put the index of the ID from IDholder  
 if (IDholder.contains(mu1.tokenName)) muop1+=IDholder.indexOf(mu1.tokenName); //if ad1 is an ID use index for op1  
 if (IDholder.contains(mu2.tokenName)) muop2+=IDholder.indexOf(mu2.tokenName); //if ad2 is an ID use index for op2  
 //at this point types are fine take the mulop operator and choose action  
 String mul=multok.tokenValue;  
 switch (mul){  
 case "\*":  
 resultmu = firstmu\*secondmu;  
 termMu = new Token(Integer.*toString*(resultmu),"int",Integer.*toString*(resultmu), 112, multok.tokenLine,null);  
 termMu.tokenTempCode=codeGenCounter; //assin counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("term",termMu);  
 quadPrint("MUL ",muop1,muop2,String.*valueOf*(termMu.tokenTempCode)); //how to get token being assigned to?  
 return termMu.tokenValue;  
 case "/":  
 if (mu2.tokenValue.equals("0")) Error.*DivideByZero*(mu2.tokenName);  
 resultmu = firstmu/secondmu;  
 termMu = new Token(Integer.*toString*(resultmu),"int",Integer.*toString*(resultmu), 112, multok.tokenLine,null);  
 termMu.tokenTempCode=codeGenCounter; //assin counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("term",termMu);  
 quadPrint("DIV ",muop1,muop2,String.*valueOf*(termMu.tokenTempCode));  
 return termMu.tokenValue;  
 case "%":  
 resultmu = firstmu%secondmu;  
 termMu = new Token(Integer.*toString*(resultmu),"int",Integer.*toString*(resultmu), 112, multok.tokenLine,null);  
 termMu.tokenTempCode=codeGenCounter; //assin counter to token  
 codeGenCounter++; //increment counter  
 symbolTable.put("term",termMu);  
 quadPrint("MOD ",muop1,muop2,String.*valueOf*(termMu.tokenTempCode));  
 return termMu.tokenValue;  
 default:  
 System.*out*.println("ERROR: Invalid mulop operator parsed at line: "+multok.tokenLine); //error test  
 }  
 }catch (NumberFormatException e){  
 System.*out*.println("Illegal Input. At line: "+multok.tokenLine); //error  
 e.printStackTrace();  
 System.*exit*(1);  
 }   
 break;

Figure 7: Multiplication-Division-Modulo Cases in Parser class

## Control Statements

The code shown below demonstrates the use of the if condition, get() method and put() method.

### If Condition Statement

case 18://{ 107(statement), 5, 107(statement), 5( ) ), 110(expression), 4( ( ), 19(if)}  
 if (backPatchIfFlag) backPatching();  
 break;

Figure 8: The If statement

It can be noted that all that the semantic case for the if case contains but a call to the back patching method shown here as long as the boolean flag was set to true:

public void backPatching(){  
 int backNum = backStack.pop(); //get the number to update op3 of JNE backpatch line from stack  
 ArrayList<String> quad = *printlist*.get(backNum);   
 quad.remove(3); //remove the third operand  
 quad.add(String.*valueOf*("#"+*printlist*.size())); //add the new op3 in place of what was removed  
 *printlist*.set(backNum,quad); //replace the arraylist in the index of the backNum  
 backPatchIfFlag=false; //set flag to back to false  
}

Figure 9: Backpatching

The boolean flag is set at the start of the Scanning method in which when a token with “if” as its property, the flag is set to true. The flag is again set to false after backpatching was performed.

### Get () Statement

case 16://{ 107(statement), 4, 5( ) ), 3(ID), 4( ( ), 17(get)}  
 //intermediate code ; you get input from user  
 Token ID16 = *parseStack*.get(*parseStack*.size()-4);  
 if (IDholder.contains(ID16.tokenName)) {  
 quadPrint("SYS ","#1","",""+IDholder.indexOf(ID16.tokenName));  
 }  
 break;

Figure 10: The Get() Method

### Put () Statement

case 17://{ 107(statement), 4, 5( ) ), 3(ID), 4( ( ), 18(put)},  
 //intermediate code ; put is printing data. Printing characters is different from numbers.  
 Token putt = *parseStack*.get(*parseStack*.size()-4); //get ID from stack  
 Token putMe = symbolTable.get(putt.tokenName); // ID from symboltable using putt  
 try {  
 String value = putMe.tokenValue; //value to print  
 if (putMe.tokenType.equals("int")) {   
 //quadPrint("SYS ","#-2",""+IDholder.indexOf(putMe.tokenName),"");  
 quadPrint("SYS ","#-1",""+IDholder.indexOf(putMe.tokenName),"");  
 }  
 if (putMe.tokenType.equals("char")){  
 int charValue = Integer.*valueOf*(value); //get ascii   
 //quadPrint("SYS ","#-1",""+(char)charValue,"");  
 quadPrint("SYS ","#-2","#"+value,"");  
 }  
 } catch (NullPointerException e){  
 e.printStackTrace();  
 System.*exit*(1);  
 } catch (NumberFormatException e){  
 }  
 break;

Figure 11: The Put() Method

## Basic Algorithm

COSC470\_cyaustria.java Main Algorithm(){

Call to Lexical Analyzer

Perform Lexical Analysis – clean text file of spaces and prepare Tokens

Call to Scanner – passes the tokens to Parser

Parser – parses token, passes token to Semantics before processing next token

Semantic check on token is performed, if not returns false

Code Generation – forms the quad necessary for mini nad mice

Parser dictates if it passes Parsing, if not returns false.

Call to print token on output file.

Output file is generated

}

# Error Messages

Error messages seen through parsing are as follows:

* RuleNotFound() – Means that the next action is code 99 or Error and thus parsing has failed.
* RHSNotFound() – Means that the Array for that Rule Number was not found by the compiler. This means that either the Rule is out of bounds of the Parse Table or a -1 due to no number found (-1 is a default value set to ensure fail-safe procedure.
* IDTypeMismatch() – Means that the two tokens being compared or assigned is of two different types or not found. Not to be confused with NumberFormatException().
* NumberFormatException() – Types of two values being compared are of different data types. Often an issue with ints or char values.
* DivideByZero() – Means that the divisor or second value or number is a 0 and thus should not continue execution.
* NullPointerException() – Means that there are no values being fetched from location or being assigned.
* FileNotFoundException() – Means that the file.txt was not in the program folder

# What happens when the program crashes?

The program exits with code (1) indicating that the program did not complete the Compiling process successfully. There will be a process.txt that will show the user the entire process that compiler took to get to the point before the crash. The process.txt will show the error message and if available where in the compiler the error occurred. No .obj file will be generated due to the crash.

# Appendix

## Program Example

public static void TEST(SOMEPARAMETER){  
 var NUMTEST, V1, V2, RESULT : int; //declare int  
 var CHAR\_TEST, A, B, Q, S: char; //declare char  
 {  
 NUMTEST = 1; //int assignment  
 RESULT = NUMTEST;  
 RESULT = 1 + 3 + 5 + 7;  
 V1 = 1; //char assignment  
 V2 = 1+1+V1;  
 A = 'A';  
 B = 'B';  
 Q = '?';  
 S = '\_';  
 CHAR\_TEST = 'a';   
   
 if(CHAR\_TEST >= 'A'){ //if with comparison  
 NUMTEST = 2;   
 put(S); //put space  
 if(3 > 2) put(NUMTEST) //nested if; last line of compound statement does not have semi-colon  
 }; //semi-colon here because not the end yet  
 put(S); //put space  
 put(V2);  
 put(S); //put space  
 put(A); //put on terminal 'A'  
 put(Q); //put on terminal '?'  
 put(S); //put space  
 get(V1); //get user input and put to V1  
  
 put(B); //put on terminal 'B'  
 put(Q); //put on terminal '?'  
 put(S); //put space  
 get(V2); //get user input and put to V2  
  
 RESULT = V1 + V2;  
  
 put(RESULT) //Last line does not have semi-colon  
 }  
}  
$

## Summary of Errors

### RuleNotFound()

Means that the next action is code 99 or Error and thus parsing has failed.

### RHSNotFound()

Means that the Array for that Rule Number was not found by the compiler. This means that either the Rule is out of bounds of the Parse Table or a -1 due to no number found (-1 is a default value set to ensure fail-safe procedure.

### IDTypeMismatch()

Means that the two tokens being compared or assigned is of two different types or not found. Not to be confused with NumberFormatException().

### NumberFormatException()

Types of two values being compared are of different data types. Often an issue with ints or char values.

### DivideByZero()

Means that the divisor or second value or number is a 0 and thus should not continue execution.

### NullPointerException()

Means that there are no values being fetched from location or being assigned.

### FileNotFoundException()

Means that the file.txt was not in the program folder

## Summary of Syntax

|  |
| --- |
| The last executable line of the program must NOT have a ; at the end of its line. |
| The program must begin with either public or private (access). |
| Variable declarations must come after the method line |
| Variable declarations must start with var, a space between that and the ID. |
| If declaring multiple variables, separate the variables with a comma between them. |
| The declaring the type for the variable declaration line is denoted by a space after the last ID declared followed by a : sign , a space, the type and ended with a ; |
| The ; symbol will denote that the line has ended for any lines in the program save for the last line of the program before the } } $ symbols and the special if conditional exception. |
| Variable declarations for a type must all be declared in one line. Multiple instances declaration of the variables throughout the code are not allowed. |
| The body pf the code must be in cased in { } symbols. |
| The assignment operator is denoted by the = symbol. |
| Operations that can be performed (as well as their respective symbols are as follows: |
| Addition (+) ; Subtraction (-) ; Multiplication (\*) ; Division (/) ; Modulo (%) |
| Integer assignments to variables can be directly assigned. |
| Character assignments need to be enclosed in ‘ ‘ symbols. |
| Variable IDs can also be assigned to other variable ID so long as the ID being assigned was declared and initialized. |
| Multiple operators, numbers and IDs can be assigned to a variable ID provided that they are ended with a ; symbol. |
| The reserve method in order to get input from a user is get(); |
| The reserve method in order to display data on the terminal is put(); |
| No operations are allowed between the () of either get or put. |
| In order to display a variable, the variable must first be initialized via assignment of a int or char. |
| Conditionals take the form of if () lines. |
| Case A Condition: If there is only ONE line to be executed by an if () line, then there is no need to enclose the executed line with { } symbols and the executable line must be placed next to the if () line. |
| Case B Condition: If there are more lines to be executed by an if () line, then { } symbols are necessary in order for the program not to crash and ended with a ; so long as the line with the } is not the last line of the program. See Rule 2 for more information. |
| Nested ifs are allowed provided that the nested if are only Case A Condition. |
| The program must end with a $ sign. |
| Comments are denoted by // and all characters or symbols after it are ignored. |