COSC470 Compiler Design and Implementa

User Reference Manual COSC470\_cyaustria0 Compiler v2

Catherine Austria

for Prof. Mohsen Chitsaz

08/25/2016

Table of Contents

[Introduction 3](#_Toc459920192)

[Chapters 3](#_Toc459920193)

[How to Start 3](#_Toc459920194)

[Data and Data Types 4](#_Toc459920195)

[Data 4](#_Toc459920196)

[Data Types 8](#_Toc459920197)

[Operators 8](#_Toc459920198)

[Condition Statements 9](#_Toc459920199)

[Appendixes 10](#_Toc459920200)

[Syntax Diagram 10](#_Toc459920201)

[Error Messages 11](#_Toc459920202)

**List of Figures**

Figure 1: File.txt as shown in the Compiler Folder pg.3

**List of Tables**

Table 1: Operators ………………………………………………………………………………………………………………………………….9

# Introduction

This User Manual is designed to walk a user through chapters with details of how to use the COSC470\_cyaustria0 v2 and what they can expect from this compiler. This manual will cover what is needed in order to run their programs and what will happen to their program through the process of compiling. The manual will also describe a brief look into the grammar; what kinds of data and data types that are valid as well as important operators, condition statements and loops worth knowing. This manual also contains in its appendixes various syntax diagrams and error messages that may come up during compilation. An Index is provided for any other terms and concepts worth nothing.

# Chapters

## How to Start

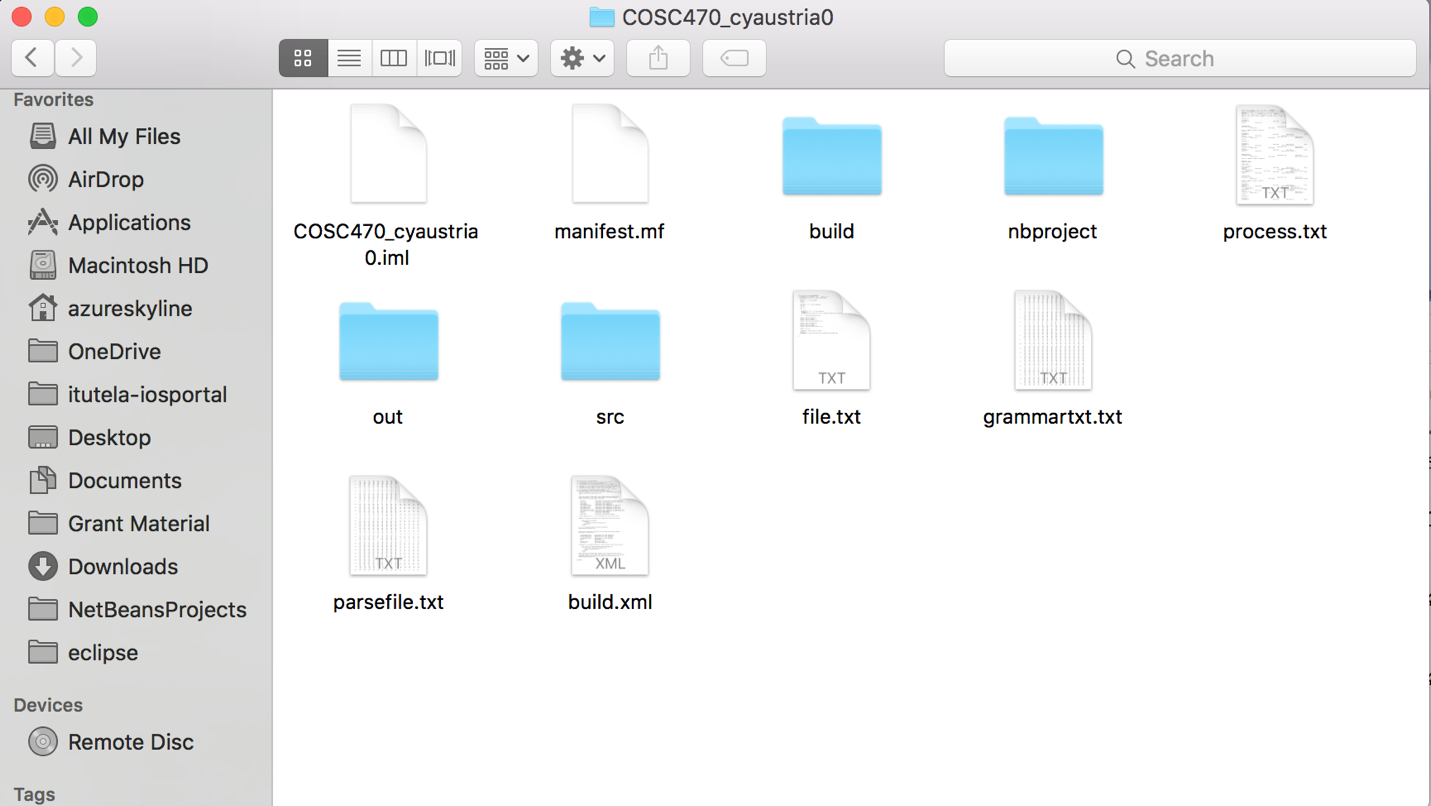
The COSC470\_cyaustria0 v2 requires that the user write their program on a text file with a .txt extension. This text file will be placed in the folder of the program as shown below. 

Figure 1: File.txt as shown in the Compiler Folder

The user can choose to compile COSC470\_cyaustria0.java and all other .java files inside the src folder through the cmd line or they can run the program through an IDE. The required .java files and their purpose are listed below:

* COSC470\_cyaustria0.java – serves as the main program
* Scanner.java – scans the file.txt file and prepares tokens for the compiler to handle
* Parser.java – performs the parsing of the tokens and contains the semantic check and code generation
* Error.java – error class keeps errors apart from those handled by java and will be discussed thoroughly later
* Token.java – This class contains the attributes necessary to the parsing process and serves as the basic unit manipulated by the compiler
* Rules.java – This class serves as the dictionary of the compiler. It contains the Parse Table and Go To table used to derive actions and elements for the use of the Parse Stack of this compiler.

The compiler will output a COSC470\_cyaustria0.obj file to be used for mini and mice.

## Data and Data Types

### Data

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.

\*It is worth noting that for every line of the user program, comments after the (;) and started by // are ignored by the compiler.

### Data Types

This compiler only handles two kinds of data types:

* Int
* Char

The data must be declared in a single line for each type and each line must start with a “var” word followed by the individual variables known to the program as IDs. These IDs must be separated with a comma in between them and must be capitalized to signify to the program that these are identifiers(IDs) token and not to be confused with other forms of tokens. The line is ended by a (☺ a space and the data type of either int or char and a (;) to signify the end of the line. Comments after the (;) will be ignored. An example for both int and char data types are shown below:

var NUM, VAR1, VAR2, SUM : int;

var CHAR1, CHAR2, ETC : char;

This compiler requires rules for assignment as well. Integer assignments must end with a (;) and Character assignments must be placed within ‘ ‘ and ende with (;). Comments after the (;) will be ignored. An example for both int and char data types are shown below:

NUM = 1;

CHAR1 = ‘?’;

SUM = VAR1 + 2;

Comparisons on the IDs are allowed. If a user would like to compare a char, they must still place the character in between ‘ ‘ to be in keeping with the grammar rules. Number can be compared as is. An example for both int and char data types are shown below:

if(CHAR1 > ‘A’){

if(2<NUM){

## Operators

This is a table of the operations that can be used:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code | Rule in Grammar Used | Token Name | Type | Sample Code |
| 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 1: Operators

## Condition Statements

Condition statements are can be written in the form below:

If (expression)

statement

Is valid:

* If is only one line to be executed after the If (expression) ; then brackets **are not needed**.
* If there are multiple lines to be executed however; brackets to enclose the multiple lines **are needed**.

# Appendixes

## Syntax Diagram

access static void ID (id\_list){ //comments are ignored due to //

var ID1, ID2 : int; // integer declaration

var ID3, ID4 : char; //character declaration

{

ID1 = 1; //number assignment

ID3 = ‘a’; //character assignment

SomeStatements;

If (expression) { //bracket needed for more than one executable lines

Statement;

MoreStatement;

If (expression) statement // if nesting if, this must be on one line

}; //

EvenMoreStatements;

statement //Last line does not have ;

}

}

$

## Error Messages

* 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