# 

# 

# 

# 

# Compiler Design for i Can Code (iCC)

# CS 153

Fall 2015

December 9, 2015

Submitted to Dr.Ronald Mak

Aishwarya Venketeswaran

Rishi Bommu Reddy

Dave Zheng

# Table of Contents

[Compiler Design for i Can Code (iCC)](#h.u64v2vlf4dxb)

[CS 153](#h.7rrhx2woa7gk)

[Table of Contents](#h.fhiclq6hzw85)

[Overview of language](#h.s1s24e1s5wil)

[Design Principles](#h.m7am3zfxlq9v)

[UML Diagrams](#h.kmseatg9bb3e)

[How to build a program in our language?](#h.hfua29e4era9)

[Primitive Types](#h.m7am3zfxlq9v)

[Basic arithmetic operations with operator precedence.](#h.yd696ohzqj9w)

[String functions](#h.kjbhqucdg38o)

[Assignment statements](#h.m7am3zfxlq9v)

[Input and Output](#h.h1g7ta6nuuu)

[Data types](#h.m7am3zfxlq9v)

[List](#h.dfln27fe511p)

[Set](#h.m9y8vox5lf6q)

[Map](#h.osxd2588bepo)

[Conditional control statement](#h.78pus7f8pwsh)

[If - Else](#h.vx3jbhj7juk6)

[Looping control statement.](#h.m7am3zfxlq9v)

[For](#h.spfymude0mo1)

[While](#h.yl17izrve8sw)

[Procedures with calls and returns.](#h.pwp1seehy3ku)

[Parameters passed by value](#h.m7am3zfxlq9v)

[Error recovery](#h.jb1aq51le17u)

[Sample Programs](#h.m7am3zfxlq9v)

[Bubble Sort](#h.bta0eaujb763)

[Number of occurrences of letters](#h.ctcxnhrg52mf)

[Game](#h.ozoihqrygs64)

[Errors](#h.x8usnr68ja69)

# 

# Overview of language

The programming language of choice is called iCC or (i Can Code) which allows those new to the programming scene to learn the ropes of programming without having to worry too much about syntax, parsing errors and all the other kinks that may turn away potential students. We wanted the best features from all our favorite languages that had elegant yet simple features: Java, Pascal, Python, Ruby. Elements taken from all these prominent languages are used to allow for a better introduction into the programming language. The following are the specifications for our language, along with notes on the extra work put in to ensure that the language is robust enough to handle the needs of a budding developer.

This language was built using software engineering principles of building off code that works. Github was the tool of choice for version control with Trello being the mode of tracking issues. In the future, iCC can be built further from where it was left off, adding endless possibilities from object-oriented programming to multithreaded support. The language is parsed with JavaCC, with assembly backend built with the Java Jasmin assembly language.

# Design Principles

As mentioned earlier, iCC was built on using good software engineering methodology which is to start on small working portions of code and building up. The front end, which is the parser and scanner, were built first using JavaCC to run .jj file extensions that would auto-generate Java files for processing. The intermediate tier would be the next to be built and finally the backend which would walk the parse tree (abstract syntax tree) and generate a .j file extension for Jasmin code. The Jasmin assemble would then assemble the assembly language into a .class file which would be run by the Java Virtual Machine as if it were a Java file.

iCC as of now does not have a particular file extension, instead choosing to stay at a .txt file. This can be seen as a way to further simplify the coding process and take away the intimidation that languages like Java has when it introduces difficult concepts such as “System.out.println()” for printing.

**At least two data types with type checking.**

* List, Set, Map

**Basic arithmetic operations with operator precedence.**

* +, -, --, ++, +=, -=, \*, /
* Syntax diagrams properly allow for operator precedence derived from Pascal

**Assignment statements.**

* int, double, float, boolean

**At least one conditional control statement (e.g., IF).**

* if, else, else if

**At least one looping control statement.**

* while, for, nested while/for

**Procedures or functions with calls and returns.**

* Supported

**Parameters passed by value or by reference.**

* Supported

**Basic error recovery.**

* Recursive error handling

**“Nontrivial” sample programs written in the source language.**

* BubbleSort
* Occurrences of letters in a list using a HashMap
* A game!

**Generate Jasmin code that can be assembled.**

* Supported in CodeGenerator.java

**Execute the resulting .class file standalone (preferred) or with a test harness.**

* With standalone class
  + java -cp jasmin.jar;PascalRTL.jar;DataStruct.jar;.;\_\_\_\_\_\_.class

**No crashes (e.g., null pointer exceptions).**

* Taken care of with error handling!

**Input Read With Command Line**

# **Syntax Diagrams**

Syntax diagrams can be found on syntax\_diagram.xhtml for a complete listing. Generations are a courtesy from <http://www.bottlecaps.de/rr/ui>

# 

# UML Diagrams

Intermediate Tier: Abstract Syntax Tree Generation

# 

# 

# How to build a program in our language?

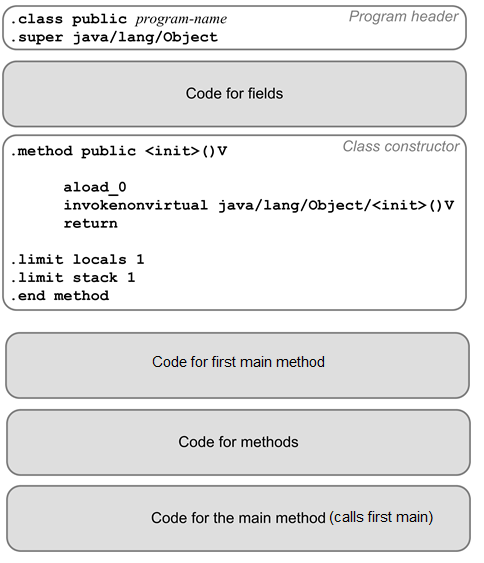
JavaCC generates a Java file called ExprParser.java which runs the front, middle and backend of the program. A Linux shell of Windows command line can be used to generate the entire front end and middle tier. The InputFile directory contains all of the files needed to run various sample programs, while the OutputFile directory has the generated code in the Jasmin assembly language. Any code written in iCC is stored in the InputFile directory and taken out when needed to be parsed and assembled.

**See README.txt for details.**

|  |
| --- |
| java -jar OutputFiles/jasmin.jar OutputFiles/InputFiles/$1.j java -cp '.:OutputFiles/jasmin.jar:OutputFiles/PascalRTL.jar:OutputFiles/DataStruct.jar' $1 mv $1.class OutputFiles/$1.class |

The shell commands then generate a .class file which is run by the Java Virtual Machine which cannot tell if the code originated from a Java file or not.

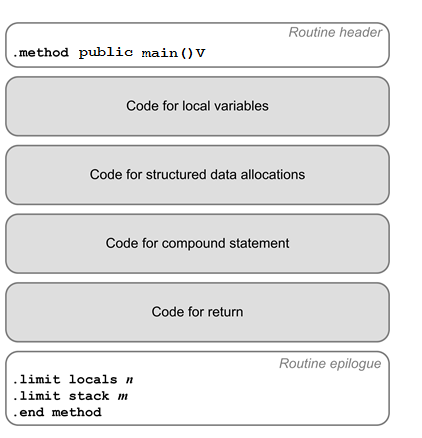
**General Code Template for iCC Programs**



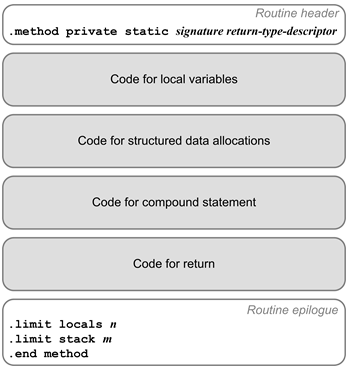
**Code for Fields**

|  |
| --- |
| .field private static \_runTimer LRunTimer;  .field private static \_standardIn LPascalTextIn; |

**Code for First Main Method**



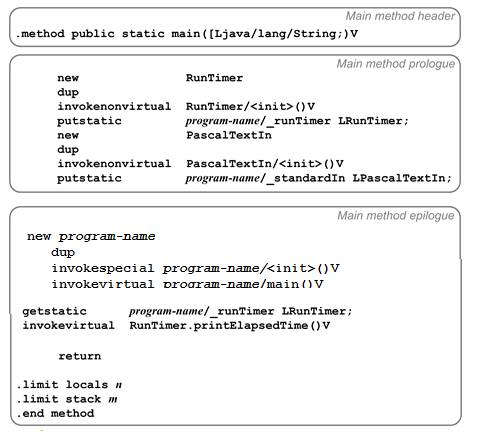
**Code for Methods**



**Code for Main Method**

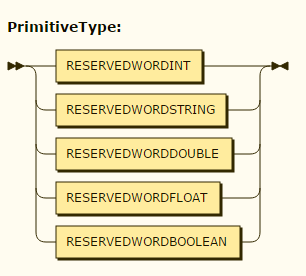
This method calls the initial main method, which does the heart of the work involved in our code.

main()V:



# Primitive Types

A variety of primitive types are supported by iCC. These types are int, boolean, String and double. The integer operations have operator precedence based on the way that the front end was coded through the EBNF form.

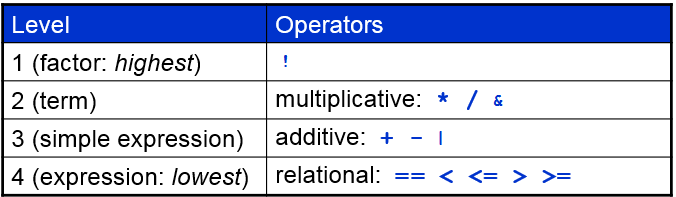


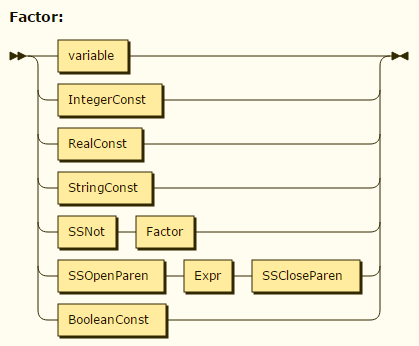
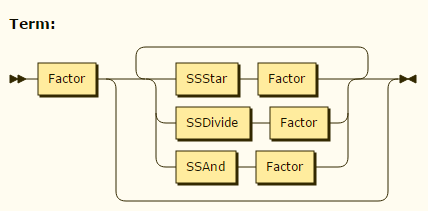
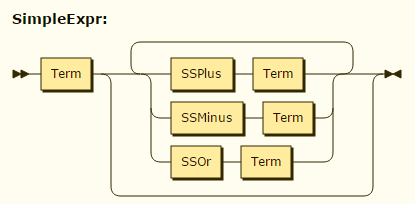
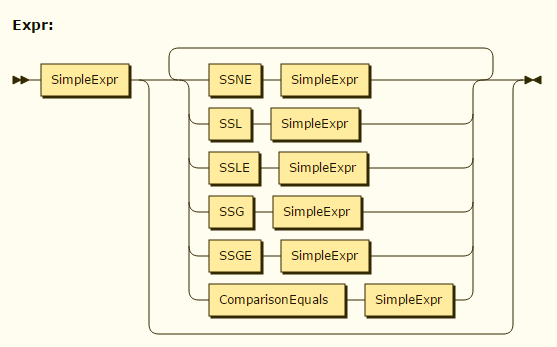
# 

# 

# Basic arithmetic operations with operator precedence.

Operator precedence is maintained by proper nesting of Expressions, Simple Expressions, Term and Factor. It follows very closely to how Pascal orders its operator precedence:





# 

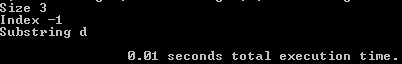
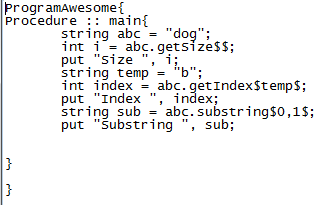
# 

# 

# String functions

Program File: TestString.txt

The following code snippet demonstrates string operations.



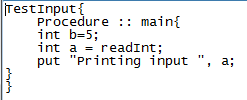
# 

# 

# Assignment statements

# 

Program File: TestInput.txt



Assignment statements work like they do in Java where the declaration and assignments are strongly typed. The types as mentioned before are the primitive types along with more advanced data types such as HashMap, List and Set.

# 

# 

# 

# Input and Output

# We have an awesome game you can play in the terminal of your computer. It is called Game.txt. The game uses the core functions of our language aided with input and output functionality to give you an entertaining experience. The game uses our Languages capability to take in input as numbers from the users. This allows us to interpret different actions that the user wants to do. This allows our user to make different choices in the game and have fun with the experience.

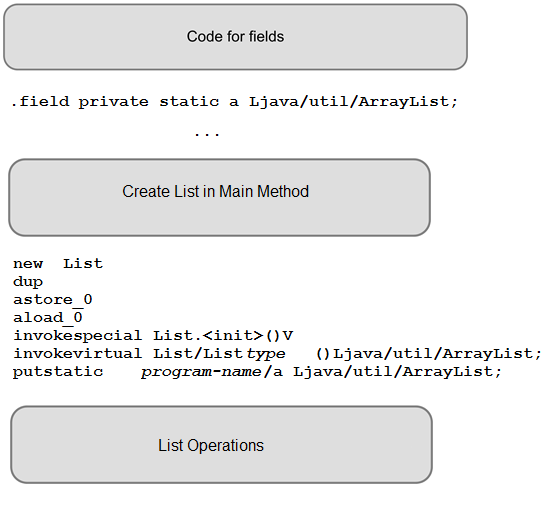
# Input functionality in our program comes from calling the **readInt** Declaration while asks the user for input. This can then be set to any integer variable in the program. This allows manipulation in control within the program as well as the capability to ask for multiple input.

# Output in our program was designed for simplicity. We wanted the use of a print statement that worked with all types of variables and declarations. Output is done in our language through the **put** command. This command allows users to print multiple variables and strings with ease. We can use the put command with various inputs so that we can have a wide array of choices for the user.

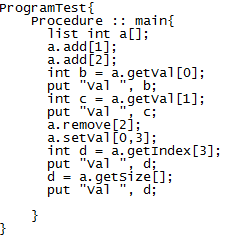
# 

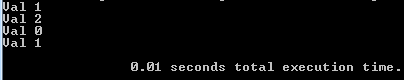
# Data types

## List

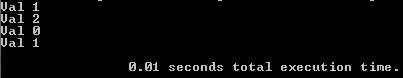
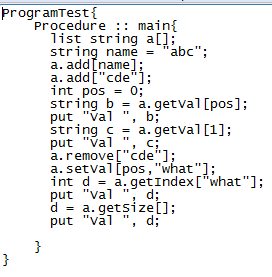


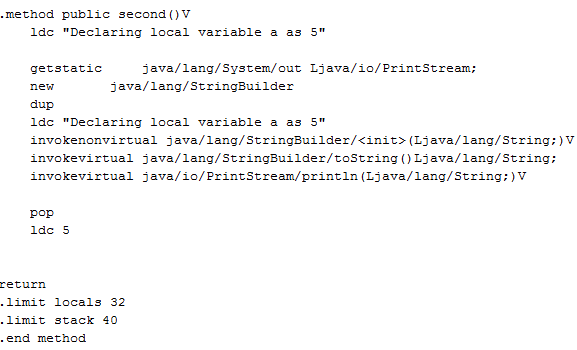
Program File: TestListInteger.txt



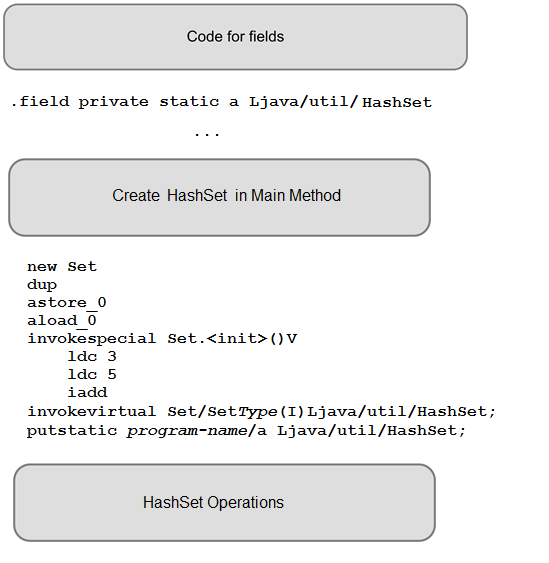


Program File: TestListString.txt

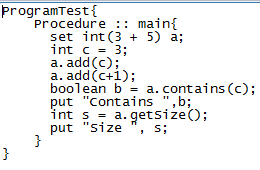




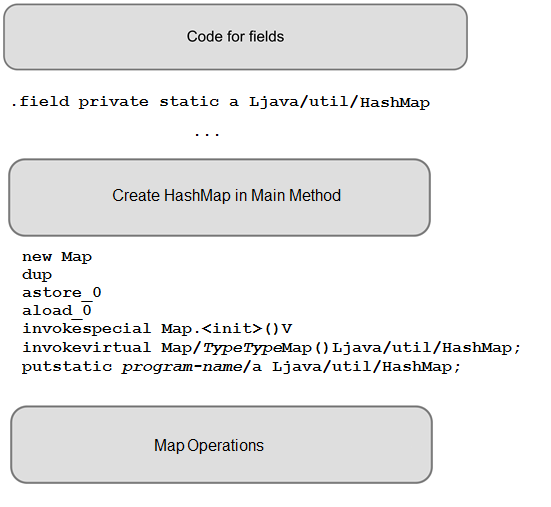
## Set



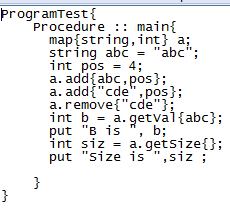
Program File: TestSet.txt



## Map



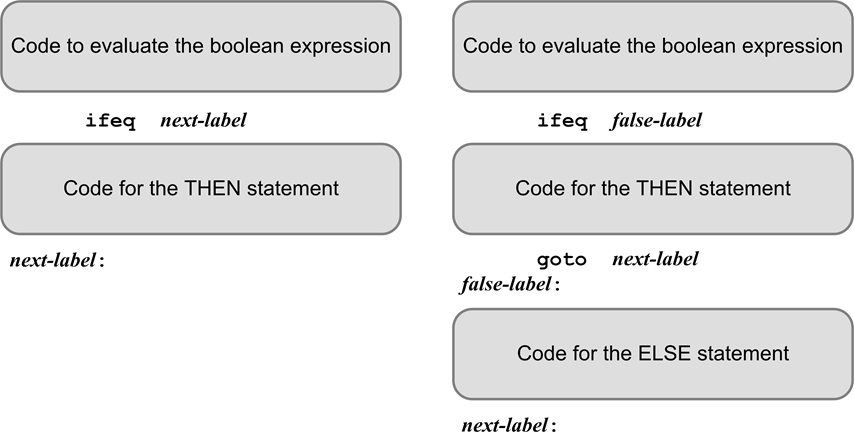
Program File: TestMap.txt



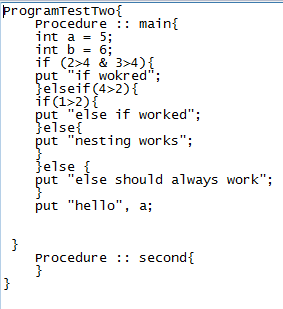
# 

# 

# Conditional control statement

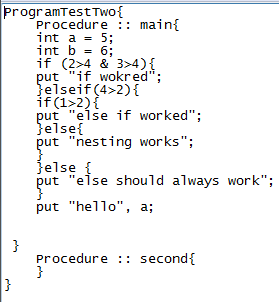


Program File: TestIf.txt



## If - Else

Program File: TestIf.txt

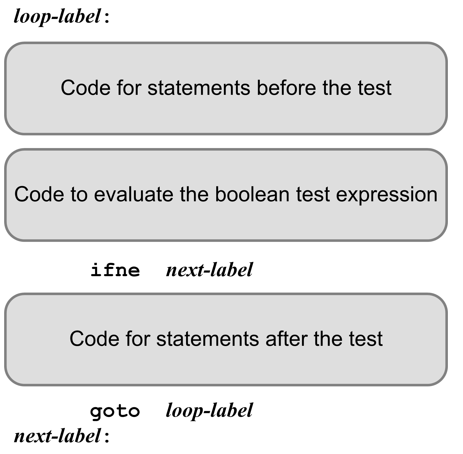


# 

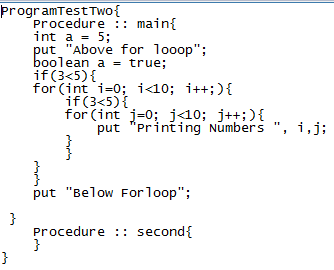
# 

# Looping control statement.

## For



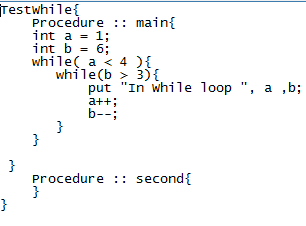
Program File: TestFor.txt



TestFor.txt prints 100 numbers.

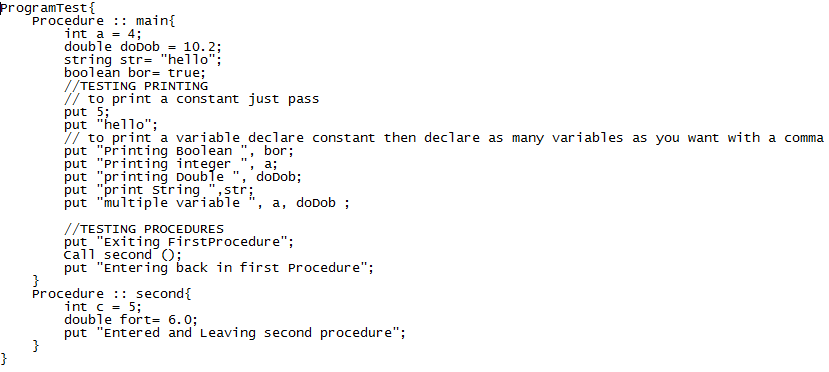
## While

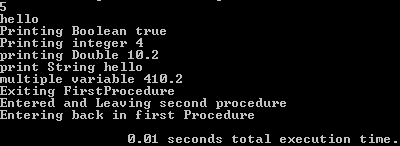
Program File: TestWhile.txt



# Procedures with calls and returns.

Program File: TestProc.txt





# Parameters passed by value

# 

# 

# 

# 

# Error recovery

See TestError.txt, TestProgramStructure.txt, TestTypeChecking.txt parsing shows how we handle various types of errors. The program will automatically be able to process the errors and terminate without reaching any issues.

The TestError.txt will be used to handle basic errors. It’s main purpose is handle general purpose errors within the program. The TestProgramStructure.txt is our main structure handler. If the text file that is inputted is complete garbage, our program will make sure that this does not return a error. TestTypeChecking will assure that we do not assign incorrect types. For example you do not want to declare a boolean as a string and return valid output. To assure that we have accurate error recovery, we have these three test files that show how we handle errors. All three of these files do not have .j or .class files due to the errors they print out.

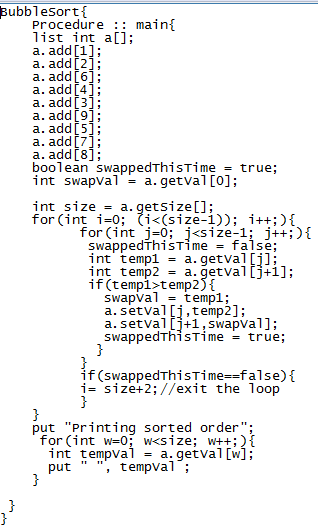
The Below line will be printed whenever you have errors, Indicating that a .j file has not been generated for you. This will assure that you are not trying to run a garbage code on your local computer. ->

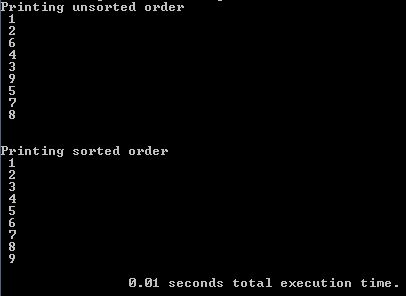
Error(s) have been thrown, please fix and recompile

# Sample Programs

## Bubble Sort

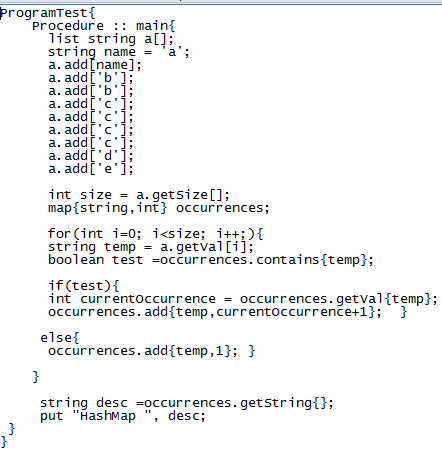
Program File: BubbleSort.txt





## Number of occurrences of letters

Program File: SampleProgram.txt





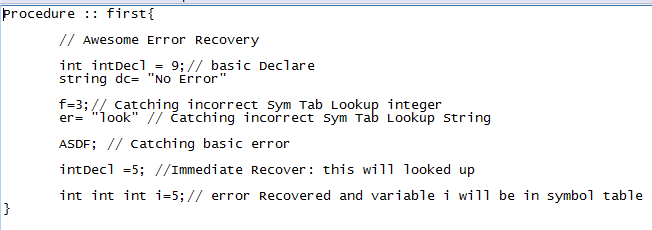
## Game

Game.txt is a mini game with plenty of if statements to keep the fun continuing.

|  |
| --- |
| TheGreatGame{  Procedure :: declareVariables{  int input=0;  string weaponOne="";  string weaponTwo="";  string classChoice="";  }  Procedure :: main{  boolean continueGame=true;  while(continueGame==true){  Call StartJourney;  Call BattleMonster;  Call PlayPredict;  put "You have been enlightened physically and mentally";  put "Do you want to reach enlightenment with another Class or quit";  put "1 Play Again with same or new Hero";  put "2 quit";  Call inputInt;  if(input!=1){  continueGame=false;  }  }    }  Procedure :: PlayPredict{  put "You have reached the House of a Fortune Teller";  put "You go in and the fortune teller agrees to train you in prediction";  put "The fortune teller says I have picked a number from one to a thousand";  put "Can you guess my number";  boolean foundNumber = false;  while(foundNumber==false){  Call inputInt;  if(input==276){  put "You have predicted the fortune tellers number very quickly";  put "You are enlightened in the field of mentalism";  put "";  put "";  foundNumber=true;  }elseif(input>400){  put "You have guessed too high";  put "Guess Again and learn to Become one with nature";  }elseif(input<200){  put "You have guessed too low";  put "Guess Again and feel the force";  }elseif(input>276){  put "You have guessed high but you are close";  put "guess again and a clue that one of the numbers is a 7";  }elseif(input<276){  put "You have guessed low but you are close";  put "Guess Again and one of the numbers is a 6";  }  }  }  Procedure :: BattleMonster{  put "You have started your journey ",classChoice;  boolean success=false;  put "A Wild Centaur Appears and is About to Attack";  put "What will you do";    while(success!=true){  put "1 Attack with ", weaponOne;  put "2 Attack with ", weaponTwo;  put "3 Try to Reason with the beast";  Call inputInt;  if(input==1){  put "The Centaur Dodged your ", weaponOne;  }elseif(input==2){  put "You have killed the beast and continue your journey";  put "You are enlightened in the field of physical defense and offense";  put "";  success=true;  }elseif(input==3){  put "The Beast has stopped and argues Kantian ethics";  put "The Beast sees its impossible to live if he universally does not attack you";  put "Thus he must still commence battle";  }  }  }  Procedure :: StartJourney{  put "You have started on a journey to enlightenment";  put "Your journey starts with you choosing your class";  put "Do you want to be a";  put "1 Mage";  put "2 Warrior";  put "3 Thief";  Call declareVariables;  Call inputInt;  if(input==1){  put "You have chosen the Mage";  Call CreateMage;  }elseif(input==2){  put "You have chosen Warrior";  Call CreateWarrior;  }elseif(input==3){  put "You have chosen Thief";  Call CreateThief;  }  }  Procedure :: CreateWarrior{  classChoice= "Warrior";  weaponOne="Battle Axe";  weaponTwo="Great Sword";    }  Procedure :: CreateMage{  classChoice= "Mage";  weaponOne="Fire Bolt";  weaponTwo="Frost Bolt";  }  Procedure :: CreateThief{  classChoice= "Thief";  weaponOne="Dagger";  weaponTwo="Long Bow";  }  Procedure :: inputInt{  put "Please enter a valid Number";  input = readInt;  put "";  put "";  put "";  put "CONTINUED HERE";    }  } |

## Errors

Program File: TestError.txt



|  |
| --- |
| run:  Type Check Error This is not a Integer  Type Check Error This is not a String  Type Check Error This is not a Double  \*\*\* ERROR: Line 17 after ";"  \*\*\* ERROR: Line 18 after "int"  ===== CROSS-REFERENCE TABLE =====  \*\*\* PROGRAM Program \*\*\*  Identifier Line numbers Type specification  ---------- ------------ ------------------  asd 016  Defined as: variable  Scope nesting level: 1  Type form = enumeration, Type id = double  dc 007  Defined as: variable  Scope nesting level: 1  Type form = scalar, Type id = string  i 018  Defined as: variable  Scope nesting level: 1  Type form = scalar, Type id = integer  intDecl 013  Defined as: variable  Scope nesting level: 1  Type form = enumeration, Type id = float  s 017  Defined as: variable  Scope nesting level: 1  Type form = scalar, Type id = string  ===== INTERMEDIATE CODE =====  \*\*\* PROGRAM Program \*\*\*  <Program>  <Procedure value="first">  <Commands>  <Statement>  <Declaration type\_id="integer">  <variableDecl id="intDecl" level="1" index="0"  type\_id="integer" />  <IntegerConst value="9" type\_id="integer" />  </Declaration>  </Statement>  <Statement>  <Declaration type\_id="string">  <variableDecl id="dc" level="1" index="1" type\_id="string"  />  <StringConst value="No Error" type\_id="string" />  </Declaration>  </Statement>  <Statement>  <Declaration type\_id="float">  <variableDecl id="intDecl" level="1" index="2"  type\_id="float" />  <IntegerConst value="5" type\_id="integer" />  </Declaration>  </Statement>  <Statement>  <Declaration type\_id="double">  <variableDecl id="asd" level="1" index="3" type\_id="double"  />  <StringConst value="Hello" type\_id="string" />  </Declaration>  </Statement>  <Statement>  <Declaration type\_id="string">  <variableDecl id="s" level="1" index="4" type\_id="string" />  <DoubleConst value="5.2" type\_id="double" />  </Declaration>  </Statement>  <handleError />  <Commands>  <handleError />  <Commands>  <Statement>  <Declaration type\_id="integer">  <variableDecl id="i" level="1" index="5"  type\_id="integer" />  <IntegerConst value="5" type\_id="integer" />  </Declaration>  </Statement>  </Commands>  </Commands>  </Commands>  </Procedure>  </Program>  Error(s) have been thrown, please fix and recompile |