COSC 461, COSC 561 **Bowie State Univ.**
Summer-I 2021

Project: Phase IV

Undergraduates

In this last phase, you will be cleaning up your compiler, in order to make it functional and user friendly. This involves modifying your compiler to generating a listing file, modifying your compiler to allow the user to specify the source, target, and listing files as command line arguments to the compiler, and writing a batch file to compile, view error messages, and assemble the output of your compiler, and a batch file to set up the environment and run the output of the assembler.

Listing.

Your compiler should list out error messages to *stderr*. Any error message should contain the line number where it was first detected. Specifically, error messages should be of the following form:

<errorMessage> ::= <lineNo> : <errorString>

Try to make your error messages descriptive enough so that you, as a SIMPL programmer, would find them useful for debugging.

Run-time Support.

Rather than compiling I/O operations straight into assembly code, I/O operations are assembled into calls to I/O routines. These I/O routines are in the file <install-path>\ simpl\lib\classes.zip, where <install-path> is the path to your SIMPL instalation folder.

To enable the Java dynamic linker to find I/O routines , they must be listed under the "CLASSPATH" environment variable. The location of the I/O routines, <install-path>\simpl\lib\classes.zip must be listed under "CLASSPATH", as well as the location of the SIMPL target file, "." (the current directory).

Batch Files and Command Line Arguments.

Your compiler should allow you to specify a source, destination, and listing file on the command line. The compiler must assign these run time arguments to *System.in*, *System.out*, and *System.err*, and then begin processing. To do this, you may find the *System.setIn()*, *System.setOut()*, and *System.setErr()* functions useful. For example, the call

```
System.setIn(new FileInputStream("glob"));
```

makes the file *glob* the input stream *System.in*. If you are working in *C*, the same operation as above can be performed with the statement

```
freopen("glob", "r", stdin);
```

Recall that the *.class* directive written at the top of a *Jasmin* target file must specify the name of the file. In SIMPL, the out-variable name should match the name of the file, and so you just used the out-variable name in Phase III. In this phase, the compiler must be modified so that it checks that the out-variable name matches the actual file name.

You need to create a batch file to run your compiler, and then run the *Jasmin* assembler on the compiled SIMPL program, and a batch file to set up the environment and run an assembled SIMPL program. Here is some information you may find useful for writing your batch files. The command to run the assembler, on a file called "*source*.j", is "jasmin -verify *source*.j ". The assembler produces output to the file "*source*.class". The command to run the JVM emulator, on a file called "*target*.class", is

"java *target* ". To run the assembler and the JVM emulator in this fashion, it is required that their locations be listed under the "PATH" environment variable. The locations of the assembler is <install-path>\simpl\bin".

You may also need to know how to access the command line arguments in a batch file. They are accessed as %1, %2, %3, etc.

Turn in a listing of your completed compiler, and a listing of your batch files. Make sure that you can compile, and run correct SIMPL programs. You will be demonstrating this ability to me.

Graduates

Requirements for the graduates are almost the same as for the undergraduates. The graduate batch file to run the compiler should run the compiler and then use *errview* to display any error messages. The *notSo* I/O library is located at <install-path>\notso\lib\ classes.zip". This file needs to be included in the "CLASSPATH" environment variable.

Errview

Errview is a program that allows you to view error messages, in the above specified format, and source code simultaneously, through a graphical interface. To run *errview*, use the command "java errview *source errList*", where *source* is the file name of the source file, including extension, and *errList* is the name of the file containing the error messages. *Errview* will display a window containing a source code section, and an error list. Double clicking on a particular error message will highlight the appropriate source line. To run *errview*, the environment variable "PATH" must be set up as described above, and the variable "CLASSPATH" must include the library <install-path>\simpl\lib\ errview.zip.

The SIMPL and *notSo* I/O routines

Because SIMPL I/O uses a graphical interface, the SIMPLE I/O routines are a little hard to understand, without a thorough understanding of the *Java* API. Nonetheless, they are given below, as an enticement to learn about *javax.swing*.

The major procedures are:

registerIO(*name*) - creates the I/O window, and displays the program name.

inputNum(*name*) - prompts with a variable name, and waits for input.

outputNum() - prints out the output variable value.

NotSo I/O is so similar that a listing of the routines is not included. Two additional routines are included in the interface:

inputLog(*name*) – prompts with a variable name, and waits for a Boolean value.

outputLog() – prints out a Boolean output variable.

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

// the main program for the SIMPL compiler
public class simpl extends Frame implements ActionListener{
    private static simpl ours;
    private String prog;
    private static final String seperator = "   ";
    private Label beg, end;
    private TextField inBox;
    private int inResult;

    // input and output procedures
    public static void registerIO(String name){
```

```
  ours = new simpl(name);
} // end of register
public simpl(String nm){
  super(scanner.BANNER);
  addWindowListener(new frameHandler());
  setLayout(new FlowLayout(FlowLayout.CENTER));
  setLocation(100, 100);
  setVisible(true);
  setSize(500,75);
  prog = nm + "(";
  add(beg = new Label(prog));
  add(end = new Label(")"));
  inResult = -1;
} // end of constructor
```

```
    public static int inputNum(String nm){
      int done;
      ours.putUpIn(nm);
      while((done = ours.getIn()) < 0)
          Thread.currentThread().yield();
      return(done);
    } // end of inputNum
    public void putUpIn(String nm){
      remove(end);
      remove(beg);
      prog = prog + seperator + nm + ":";
      beg.setText(prog);
      add(beg);
      add(inBox = new TextField("0", 5));
      inBox.selectAll();
      inBox.addActionListener(this);
      add(end);
      doLayout();
      inBox.requestFocus();
      inResult = -1;
    } // end of inputNum
    public int getIn(){
      return(inResult);
    } // end of getIn
    public void actionPerformed(ActionEvent e){
      try{
          int res = Integer.parseInt(inBox.getText());
          if(res < 0) throw new NumberFormatException(
            " $bad_integer_format$ ");
          remove(end);
          prog = prog + inBox.getText();
          remove(inBox);
          remove(beg);
          beg.setText(prog);
          add(beg);
          add(end);
          doLayout();
          inResult = res;
      }catch(RuntimeException ex){
          remove(beg);
          remove(end);
          remove(inBox);
          doLayout();
          throw ex;
      } // end of error
    } // end of action handler
    public static void outputNum(int i){
      ours.putUpOut(i);
    } // end of outputNum
    public void putUpOut(int i){
      end.setText("):" + Integer.toString(i));
      doLayout();
    } // end of putUpOut
} // end of class SIMPL
```

```
// a frame handler
class frameHandler extends WindowAdapter{
    public void windowClosing(WindowEvent e){
      e.getWindow().dispose();
      System.exit(0);
    } // end of WindowClosed
} // end of frameHandler
```