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

Project: Phase III

Undergraduate: You are to write a semantic analyzer and code generator for SIMPL. You will incorporate this with your parser from Phase II, to produce *Jasmin* assembly language for the JVM. You should write a driver program which calls the parser, then runs the semantic analyzer and the code generator on the syntax tree. To design your semantic analyzer you must first design an attribute grammar for the SIMPL syntax tree. You will turn in your attribute grammar along with your listing, and demonstrate your program

You will be using standard code templates to generate your code. These templates are based on several conventions. SIMPL programs contain expressions. All expressions in SIMPL will be evaluated in a uniform way. An expression is an operator with several operands, which are themselves expressions. By convention, to evaluate an operator, its operands are first evaluated. As each operand is evaluated it pushes its result on to the system stack. Then, the operator pops the operand values off of the stack, processes them, and pushes its result back onto the stack. In this way no temporary variables are required. All temporary storage is allocated off of the stack.

You will be compiling a SIMPL program to a single procedure (method). We follow the *Java* conventions for function calls. That is to say, arguments are pushed onto the stack, and values are returned on the stack. On the JVM, each method has its own stack, and its own local variables. The *Jasmin* assembler must be told the number of local variables each method is using, and the size of its stack. Local variables are identified by their numbers. All SIMPL variables will be stored in local variables. Your compiler must, therefore, generate labels for use with gotos, and numbers to be used as variables.

If a SIMPL paragraph declares n variables, the corresponding *Jasmin* method should allocate n+1 local variables. Local variable 0 is special purpose, and should not be used by the method. Local variables 1 through n would be used to implement the SIMPL variables. A method's stack size should be set to one more than the number of leaf nodes contained in the program body. It can be shown that this size exceeds the maximum number of items a SIMPL program can push.

It is the job of the semantic analyzer to check for static semantic errors, and calculate attributes necessary for code generation. Minimally, you are going to have to compute the following attributes:

- *SymbolTable*
- *stackSize*: the number of leaf nodes in a paragraph body.
- *numLocals*: the number of local variables in a paragraph.
- *label1*: a label used for control operators.
- *label2*: also used for control operators.

For SIMPL, the symbol table can be global. A symbol entry must contain enough information for type checking and code generation. Each entry must, minimally, include the attributes:

- *name*
- *ioType*: whether the variable is the out-variable, an in-variable, or a local variable.
- *varNum*

You can implement your own symbol table, or use the symbol table implemented in the class *SymTab*. This class is included in your SIMPL distribution.

You will write an attribute grammar for the syntax tree, and turn this in, along with your program listing. The semantic analyzer and code generator for SIMPL is

described in the SIMPL documentation. This description includes the symbol table format, code templates, and a node attribute list.

Your code generator will output to standard output, and print out error messages to standard error. The only semantic errors in SIMPL are redefining a variable, and using an undeclared variable. To handle these errors, print an error message, and abort.

Code templates for SIMPL are given in the SIMPL documentation. Examining the code templates, you will notice that the compiler must generate a sequence of labels and a sequence of local variable numbers. Local variable numbers can be generated using a counter, which is incremented each time a local variable number is allocated. A sequence of control labels, *L00001*, *L00002*, *L00003*, …, can be generated also with a counter. These labels are typically stored as strings. An alternative to storing them as strings, which requires less space, is to store only the label number as an integer. Then, the code generator could print the label using code similar to the following:

```
NumberFormat nf = NumberFormat.getNumberInstance();
nf.setMinimumIntegerDigits(5);
System.out.println("L" + nf.format((long)node.label1) + ":");
```

Graduate: As with the undergraduate students, you will write an attribute grammar for the syntax tree, and turn this in, along with your program listing. The semantic analyzer and code generator for *notSo* is described in the *notSo* documentation. This description includes the symbol table format, code templates, and a node attribute list.

For *notSo*, a symbol table entry must, minimally, include the attributes:

- *name*
- *ioType*:
- *dType*
- *varNum*

The minimal set of required node attributes is as follows:

- *symbolTable*
- *stackSize*
- *numLocals*
- *label1*
- *label2*
- *dType*: a node's data type.
- *errorOccured*: indicates an error occurred in the syntax tree, somewhere.

Semantic errors are listed below. You should handle the errors as described:

- *Child is of wrong data type*: The *dType* of the parent should be set to *err*.
- *Variable name is redeclared*: Do not enter the variable in the symbol table a second time.
- *Node is an undeclared identifier*: Set the data type to *err*.
- *Type mismatch on assignment*: Ignore this fact and continue.
- *Non-Boolean condition on* if *or* while: Ignore this fact and continue.

If any one of the errors listed is encountered, the semantic analyzer should print an error message.

When computing the *dType* attribute, if the *dType* of a child is *err*, the *dType* of the parent should also be *err*. This is not an error, and no message should be printed. Any nodes of type *ERROR* encountered during semantic analysis should have a data type of *err*, should be considered terminal nodes, and should not cause an error message to be printed. At the end of semantic analysis, if any errors occurred, or the tree contains *ERROR* nodes, the code generation phase should be canceled.

***Jasmin* and the JVM**

A JVM executable program consists of one or more *.class* files. Each class file is roughly equivalent to an object file. On the JVM linking is dynamic. This means that if a method is called which is external to the *.class* file, the method is located during run-time. This eliminates the need for a linker, which performs static linking. To narrow down the search required for the dynamic link, the environment variable, CLASSPATH, contains the paths to all library directories.

The JVM is a stack based machine. All instructions use the current method's stack to perform operations.

Below is a list of the *Jasmin* directives and JVM instructions used in the code templates.

Directives

.class public *filename*
- Specifies name of the *.class* file.

.super *superclassFile*
- The superclass of the class (like an extend clause in *Java*).

.method public static *name*(*arguments*)*type*
- Tells the assembler this is the beginning of a new method.

.end method
- Tells the assembler you are ending the current method.

.limit stack *n*
- Informs the assembler of the stack size for the current method.

.limit locals *n*
-Informs the assembler of the number of local variables (the local variables are numbered 0 to *n*-1)

Types

I	- Integer
V	- Void
Z	- Byte
Ljava/lang/String;	- Character String

Instructions

invokestatic *classFile/method*
- Call a procedure, dynamically linking to it, and popping its arguments.

return
- Return from a procedure, pushing the return value.

pop
- Pop the top element off the stack.

pop2
- Pop the top two elements off the stack.

sipush *constant*
- Push a short integer constant.

bipush *constant*
- Push a byte constant

istore *localVar*
- Pop an integer into a local variable.

iload *localVar*
- Push an integer from a local variable.

dup
- Duplicate the top value on the stack, and push the duplicate.

ifeq *label*

- Pop the stack and jump if the popped value is equal to zero.

ifne *label*
- Pop the stack and jump if the popped value is not equal to zero.

goto *label*
- Perform a jump.

ineg
- Pop the stack, negate the value popped, and push the result.

iadd
- Pop the stack twice, add the two integers popped, and push the result.

isub
- Pop the stack twice, subtract the two integers popped, and push the result.

imul
- Pop the stack twice, multiply the two integers popped, and push the result.

idiv
- Pop the stack twice, divide the two integers popped, and push the result.

ior
- Pop the stack twice, bitwise OR the two integers popped, and push the result.

ixor
- Pop the stack twice, bitwise XOR the two integers popped, and push the result.

iand
- Pop the stack twice, bitwise AND the two integers popped, and push the result.

if_icmpgt *label*
- Pop the stack twice, compare the two popped values, and jump if the first is greater than the second.

if_icmplt *label*
- Pop the stack twice, compare the two popped values, and jump if the first is less than the second.

if_icmpeq *label*
- Pop the stack twice, compare the two popped values, and jump if the first is equal to the second.

ldc *value*
- push a constant on to the stack.