PP3: Semantic Analysis and Code Generation

Due Date: **May 5, Friday 11:59pm**

In the third programming project, your job is to implement a semantic analyzer and code generator for your compiler. When you confirm the source program is free from compile-time errors, you're ready to generate code!

1 Semantic Analysis

Our semantic analyzer will traverse the parse tree (AST) constructed by the parser and validate that the semantic rules of the language are being respected, printing appropriate error messages for violations. This assignment has a bit more room for design decisions than the previous assignments. Your program will be considered correct if it verifies the semantic rules and reports appropriate errors, but there are various ways to go about accomplishing this and ultimately how you structure things is up to you. The only standard is that all the test cases we provide need to pass for your submission.

Your finished submission will have a working semantic analyzer that reports all varieties of errors. Your analyzer needs to show it can handle errors related to scoping, declarations and typing.

Give yourself plenty of time to work through the issues, ask questions, and thoroughly test your project. In particular, you need to build a solid and robust foundation to ensure you will be able to complete all the tasks of semantic analysis by the final due date.

2 Semantic Rules of Decaf

Since you are about to embark upon a journey to write a semantic analyzer, you first should know the rules you need to enforce! You will want to carefully read the typing rules, identifier scoping rules, and other restrictions of Decaf as given in the specification handout. Your compiler is responsible for reporting any transgression against the semantic language rules. For each requirement given in the spec, you may want to consider what information you will need to gather to be able to check that requirement and when and where that checking is handled.

3 Error Reporting

Running a semantically correct program through your compiler should not produce any output at this stage. However, when a rule is violated, you are responsible for printing an appropriate message for each error. Your compiler should not stop after the first error, but instead continue checking the rest of the parse tree.

4 Error Messages for pp3

For this part, you will report problems with declarations to show us that you have completed the basic functionality for declarations and scoping. The errors that you are required to catch at the checkpoint are listed below.

```
*** No declaration for Function 'f' found
```

This message is used to report undeclared identifiers. This error message is used for undeclared variables and functions.

```
*** Incompatible operands: double * string
*** Incompatible operand: ! int
```

Used to report expressions with operands of inappropriate type. Assignment, arithmetic, relational, equality, and logical operators all use the same messages. The first is for binary operators, the second for unary.

```
*** Function 'Winky' expects 4 arguments but 3 given
*** Incompatible argument 2: string given, int expected
*** Incompatible argument 3: double given, int/bool/string expected
```

Used to report mismatches between actual and formal parameters in a function call. The last one is a special-case message used for incorrect argument types to the *Print* built-in function.

```
*** Test expression must have boolean type
*** break is only allowed inside a loop
*** Incompatible return: int given, void expected
```

Used to report improper statements.

5 Error recovery

You will need to determine the appropriate action for your compiler to take after an error. The goal is to report each error once and recover in such a way that few or no further errors will result from the same root cause. For example, if a variable is declared of an undeclared named type, after reporting the error, you might be flexible in the rest of the compilation in allowing that variable to be used. Assume that once the declaration is fixed, it is likely the later uses of it will be correct as well.

6 Semantic analyzer implementation

You need to implement store declarations, manage scopes, and report declaration errors. Here is a quick sketch of the tasks that need to be performed.

- Start by making sure you are completely familiar with the semantic rules of Decaf as given in the specification handout. Look through the sample files and examine for yourself what are the errors are in the "bad" files and why the good files are well-behaved.
- A design strategy we'd recommend is implementing a polymorphic *Check*() method in the ast classes and do an in-order walk of the tree, visiting and checking each node. Checking on a VarDecl might mean ensuring the type is valid and the name is unique for this scope. Checking a LogicalExpr could verify both operands are of boolean type. Checking a BreakStmt would make sure it is in the context of a loop body.

- Design your strategy for scopes. There are many possible implementations, it's your call to figure out how to proceed. Some questions to get you thinking: What information needs to be recorded with each scope and how will you represent it? What are the different kinds of scopes and do they require any special handling? Where is the scope information stored and how did nodes get access to it? How will you manage entering and exiting scopes? What connections are needed between the levels of nested scopes?
- Re-read the Decaf spec about scope visibility – all identifiers in a scope are immediately visible when the scope is entered. Note this is different than C and C++.
- Note there are two separate scopes for a function declaration: one for the parameters and one for the body.
- Once you have a scoping system in place, when a declaration is entered into scope, you can check for conflicts. And once declarations are stored and can be retrieved, you can verify that all named types used in declarations are valid.
- Add error reporting for conflicting or improper declarations and you're done with the checkpoint.
- Establishing proper behavior for type equivalence and compatibility is a critical step. Re-read the spec and take care with the issues related to inheritance and interfaces. Test this thoroughly in isolation since so much of the expression checking will rely on it working correctly.
- Be sure to take note that many errors are handled similarly (all the arithmetic expressions, for example). With a careful design, you can unify these cases and have less code to write and debug.
- Check out the pseudo base type *errorType*, which can be used to avoid cascading error reports. If you make it compatible with all other types, you can suppress further errors from the underlying problem.
- Testing, testing, and more testing. Make up lots of cases and make sure any fixes you add don't introduce regressions. Before you submit, scan the error messages and semantic rules one last time to make sure you have caught all of them.

7 Matching our output

- When a file has more than one error, the order the errors are reported is usually correlated to lexical position within the file, i.e. an error on the first line is reported before one on the second and so on. Errors on the same line are usually reported from left to right.
- We will diff your output against the solution, and perform manual checks. If your output varies slightly from the solution (e.g., you print a few errors on the same line in reverse order, you catch the same first error, but the cascading errors are different), please document.

8 Code Generation

After semantic analysis, your compiler will traverse the abstract syntax tree, stringing together the instructions for each subtree -— to assign a variable, call a function, or whatever else is needed. The instructions need to be in the form of MIPS assembly. Your finished compiler will do code generation for all of the Decaf language as well as reporting link errors.

The debugging can be intense at times since you may need to drop down and examine the MIPS assembly to sort out the errors. By the time you're done, you'll have a pretty thorough understanding of the runtime environment for Decaf programs and will even gain a little reading familiarity with MIPS assembly.

Your program needs to take decaf source code as input and output .s file which are executable on SPIM simulator. The spim executable, which actually executes the code can be found in the posted package for phase 3 (directory "/spim"). The -file argument allows you to specify the file of MIPS assembly to execute.

NOTE: You need to concatenate defs.asm to the end of your .s code because `defs.asm` contains the utility functions (e.g., Print and ReadInteger). `defs.asm` is in the posted package of the phase.

```
% cat defs.asm >>program.s
% ./spim -file program.s
```

9 Building Your Own SPIM Simulator If Necessary

The SPIM simulator executable is provided under the "spim" directory. This section provides information for compiling SPIM on your own machine if the provided executable does not work for you. The SPIM simulator source code is also available in the package of phase 3, under the "spimsource" directory.

By default, the SPIM simulator will be installed on your */usr/bin* directory. If you want to install it on a different location, please change the "BIN_DIR", "EXCEPTION_DIR" and "MAN_DIR" in the $Makefile$ to desired locations.

Then in the *spimsimualtor/spim* directory, type in the following commands to install the SPIM simulator:

```
% make
% make test
% make install
```

For the folks who have problem with *make test*, you can download the latest version of SPIM simulator from Source Forge. The SPIM source code in decaf is incompatible with (at least) newer Linux distributions, and on these distributions, the *make test* will run forever. Therefore, if you have issues with compiling and using SPIM from the provided source code, you can directly download the executable or source code for SPIM from http://spimsimulator.sourceforge.net/. Executable is provided for all major OSes on Source Forge.

The SPIM executable in PP3.zip was compiled and tested on fox servers, so it is safe to compile SPIM on fox servers. This executable is statically linked, which should work on all Linux boxes.

10 Extra Points

Students who finished the main project can further implement the register allocation for 5 extra points. To get extra credit, you need to (1) pass all given tests for phase III; (2) implement liveness analysis, coloring algorithm and register allocation in MIPS code generation. Note that students whose phase III submission does not pass all given tests for phase III cannot get any extra points.

For the extra analysis, your program needs to output the 3-address code (in any readable form) and live analysis result (the set of live variables) for each program point. Then, your program needs to perform color algorithm to assign each variable to registers. Your program needs to further output the mapping of variables to registers, and the MIPs code based on the mapping. The TA and instructor will manually grade this part.

11 Code Generator Implementation

Here is a quick sketch of a reasonable approach:

- Plot out your strategy for assigning locations to variables. A Location object is used to identify a variable's runtime memory location, i.e. which segment (stack vs. global) and the offset relative to the segment base. Every variable, be it global or local, a parameter or a temporary, will need to have an assigned location. First of all, you try to deal with local variables, global variables, and temporaries (eventually you will also support parameters and instance variables). Figure out how/when you will make the assignment. As a first step, you may want to print out each location before doing any code generation and verify all is well. If you aren't sure you have the correct locations before you move on to generating code, you're setting yourself up for trouble. Once you have assigned locations for all variables located within the stack frame, you can calculate the frame size for the entire function, and backpatch that size into BeginFunc instruction.
- The label for the main function has to be exactly the string "main" in order for Spim to start execution properly.
- Start by generating code to load constants and add support for the built-in Print so you can verify you've got this part working. Simple variables and assignment make a good next step.
- Generate instructions for arithmetic, relational, and logical operators.
- Generating code for the control structures (if/while/for) will teach you about labels and branches. Correct use of the break statement should work for exiting while and for loops.

After this, you should be able to handle any sequence of code in a single main function. Proceeding with the rest of code generation includes:

- Generating code for other function definitions isn't much different than it was for main, other than that you need to assign locations to the function parameters and figure out your strategy for assigning function labels. Our solution uses the function name prefixed with an underbar as the function label and for classes we further prefix with the class name. You're welcome to use any scheme you like as long as it works (i.e. assigns unique labels with no confusion).
- You are to add one piece of "linker"-like functionality to verify that there is a definition for the global function *main* The error reported when the program contains no main is:

```
***Linker: function 'main' not defined
```

If there is a link error, no code should be emitted.

12 Testing your Compiler

There are various test files, both correct and incorrect, that are provided for your testing pleasure in the samples directory. As for output, if the source program is correct, the output will be assembly code and final execution output. If the source program has errors, the output consists of an error message for each error.

As always, the provided samples are a limited subset and we will be testing on many more cases. It is part of your job to think about the possibilities and devise your own test cases to ferret out issues not covered by the samples. This is particularly important for the final submission.

13 Grading

This project is worth 10 points and points will be allocated for correctness. We will run your program through the given test files from the samples directory as well as other tests of our own, using $diff - w$ to compare your output to that of our solution.

14 Deliverables

Electronically submit your entire project to Blackboard. You should submit a tar.gz of the project directory. Be sure to include a brief README file.

Because we grade the submissions using scripts, it is important that everyone uses the same directory structure. The uploaded folder should contain your source code, all dependencies, and a sub-folder called 'workdir' where you should put two files 'build.sh', and 'exec.sh'. Running 'build.sh' should build your project, and running 'exec.sh <filepath>' should execute your compiler on a source code file at <filepath>', and all output should be written to the standard output stream, so that the grader can use 'exec.sh <filepath> > <outputpath>' to redirect your output into files and compare with the ground truth.