

Programming Project 3: Semantic Analysis

Due: Wednesday, July 30th at 4:00 p.m.

The Goal

In the third programming project, your job is to implement a semantic analyzer for your compiler. We're now at the penultimate phase of the front-end. If we confirm the source program is free from compile-time errors, we're ready to generate code!

Your semantic analyzer will traverse your parse tree (as many times as you like, incidentally) and validate that the semantic rules of the language are being respected, printing appropriate error messages for violations. There are many rules being verified - there is lots of type checking, for one. Also, arithmetic operations require numbers. The actual parameters in a call must be compatible with the formal parameters. Variables must be declared and can only be used in ways that are acceptable for the declared type. The test expression used in an `if` statement must evaluate to a Boolean value. In addition to type checking, there are other rules: new declarations don't conflict with earlier ones, access control on class fields, `break` statements only appear in loops, and so on. One of the more interesting and worthwhile parts of the assignment is thinking through what is the best way to report various errors, so as to most help the programmer fix the mistake and move on.

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. There is no definitive way, but there are good and bad choices. Part of your job is brainstorming your options, making thoughtful decisions, and documenting your reasons. Your finished submission will have a working semantic analyzer that reports all varieties of errors.

Given that semantic analysis is a bigger job than the scanner or parser, we suggest that you *first completely handle errors related to scoping and declarations*, because these form the foundation for the later work. Past students speak of the semantic analyzer project in hushed and reverent tones. For many, PP3 makes the first two programming projects seem like elementary school math tests. Although an expert procrastinator can build a scanner or a parser the night before, a one night shot at PP3 is not recommended. 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.

Decaf's Semantic Rules

Before you embark on your semantic analysis journey, you should know the rules! Review the typing rules, scoping rules, and other restrictions as outlined in the specification handout. Your compiler is responsible for reporting any and all transgressions. 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.

Reporting Errors

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.

You should use our provided error-reporting facility for printing error messages. `ReportError` will identify the line number and print the source text, underlining the particular section in error.

The line that follows is a message describing the problem.

```
*** Error on line 2007
    here is the offennddnig line with the troubling section underlined
                ^^^^^^^^^^^^^^
    Misspelled words are not allowed in Decaf programs!
```

If you remember back to PP1, each token's location was placed in `yy1loc` by the scanner. `yacc` then tracked the location of each symbol on the parse stack using `@1`, `@2`, etc. As part of PP2, you were storing locations with the parse tree nodes, but at the time you may not have realized the eventual purpose was for semantic analysis. Those locations are used to provide the context for the error and precisely point out where the trouble is.

Devising clear wording for all the various error situations is actually trickier than it sounds. (Think of all the confusing messages you've seen from `g++` for instance.) The best compilers have a plethora of specialized error messages, each used in a very particular situation. However, in the interest of keeping things manageable, we will adopt a small set of fairly generic error messages and use each in a variety of contexts. Our pre-defined error messages are listed in `errors.h` and described below. Your output is expected to match our wording and punctuation.

Foundation Error Messages

To start, you should report problems with declarations to show us that you have completed the basic functionality for declarations and scoping. The three errors that you should handle first are listed below. Those portions in boldface are placeholders. They should be replaced with the particulars of the problematic declaration.

```
*** Declaration of 'a' here conflicts with declaration on line 5
*** Method 'b' must match inherited type signature
```

- These are used to report a new declaration that conflicts with an existing one. The first message is the generic message, used in most contexts (e.g. class redefinition, two parameters of the same name, and so on). The second is a specialized message identifying an attempt to overload a subclass's method.

```
*** No declaration for class 'Cow' found
```

- This message is used to report undeclared identifiers. The only undeclared identifiers you have to catch for the checkpoint are use of undeclared named types in declarations, i.e. a named type used in a variable/function declaration, a class named in an extends clause, or an interface from a implements clause. This error message is also used for undeclared variables and functions, but checking for those is a task handled after the checkpoint.

Final Submission Error Messages

Once you have your foundation, go on to add all the necessary checks to ensure that semantic rules are being followed. Here is the list of error messages from the full semantic analyzer. As before, the portions in boldface are placeholders.

```
*** No declaration for function 'Binky' found
```

Used to report undeclared identifiers (classes, interfaces, functions, variables).

```
*** Class 'Cow' does not implement entire interface 'Printable'
```

Used for a class that claims to implement an interface but fails to implement one or more of the required methods.

```
*** 'this' is only valid within class scope
```

Used to report use of **this** outside class scope.

```
*** 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.

```
*** [] can only be applied to arrays
*** Array subscript must be an integer
*** Size for NewArray must be an integer
```

Used to report incorrectly formed array subscript expressions or improper use of the `NewArray` built-in.

```
*** Function 'Winky' expects 4 arguments but 3 given
*** Incompatible argument 2: string given, string[] 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.

```
*** Cow has no such field 'trunk'
*** Cow field 'spots' only accessible within class scope
```

Used to report problems with the dot operator. Field means either variable or method. The first message reports to an attempt to apply dot to a non-class type or access a non-existent field from a class. The last is used when the field exists but is not accessible in this context.

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

Used to report improper statements.

We have deliberately tried to provide blanket error messages rather than enumerate distinct messages for all the errors. For example, the "incompatible operands" message is used when the `%` operator is applied to two `strings` or when assigning `null` to a `double` variable. This helps reduce the number of different errors you need to emit.

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.

There is some guesswork about how to proceed. If you encounter a second declaration that conflicts with the first, do you keep the first and discard the second? Replace the first with the second? Is one strategy more likely to cause fewer cascading errors later? What about if you see a declaration with a mangled type or a completely undeclared variable? Should you try to guess the intended type from the surrounding context? Should you mark the variable as having an error type and use that status to suppress additional errors? How will you constrain errors from tainting the rest of the context, such as adding five operands where the first is a string?

Ideally, you want to continue checking everything else that you can, while suppressing any cascading errors that result from the first error, i.e. those errors that would most likely be fixed if the original error were fixed. A few examples might help. Consider `b[4]` where `b` is undeclared. You report about `b`, but should you also report that you can't apply brackets to a non-array type? If you assume that if `b` was supposed to have been declared of the needed array type, the second would also be fixed. What about `b[4] + 5`? Again, if you assume the missing declaration was an `int` array, this should be fine, so it is another cascading error that should be ignored. What about `b[4.5] = 10`? `b` is still undeclared, but the array subscript is also not right. Fixing the declaration of `b` won't fix the array subscript problem, so there are two errors to report. Now consider `b[4.5] = 1 + 4.0`. The error on the right side is yet another distinct error (co-mingled `int` and `double`) and fixing the left hand side will not fix that error, so there are three errors to report. The idea is that each distinct error that requires an independent action to correct gets its own error message, but those errors due to the same root cause are not re-reported.

It will come down to judgment calls on the fringe cases. It will be up to you to make decisions and document your reasoning. Think about it carefully, try experimenting with some C++/Java compilers (competitive analysis!), decide what you think is right, and explain your strategy in your `README`.

Starter files

Little has changed since the files we gave you for PP2. You will use your own `parser.y`. Remove support for switch statements and post increment operators (they are not part of the Decaf specification and were in PP2 to force you extend the grammar a little).

We've added another generic collection class for your use, the `Hashtable`. A `Hashtable` maps string keys to values. Like the `List` class you've already used, it is templated to allow you to use with all different types of values. This may come in handy for managing symbol tables and scoping information. Read the `.h` file to learn more about its facilities. We also added a number of new tests.

We made some minor changes to the previously provided code. You may view the `diff` at http://www.stanford.edu/class/cs143/faqs/pp2_to_pp3.diff (a simple list of changed files is at http://www.stanford.edu/class/cs143/faqs/pp2_to_pp3_changed.txt). If you checked out PP2 from our `svn` repository, then you can update to the PP3 code with the following command from `myth` from the root of your checked out PP2:

```
svn update
```

```
Or checkout from scratch: svn co file:///usr/class/cs143/assignments/svnrepos/pp_base
```

```
Or remotely: svn co svn+ssh://USER@myth.stanford.edu/usr/class/cs143/assignments/svnrepos/pp_base
```

```
Or just copy the files: cp -r /usr/class/cs143/assignments/pp3_base .
```

Implementation Hints

To get started, you should store declarations, manage scopes, and report declaration errors. Here is a quick sketch of the *tasks you should tackle first*:

- Design your strategy for scopes. There are many approaches, and it's your call to figure out how to proceed. Think about what information needs to be recorded with each scope and how to represent it, think about the different kinds of scopes and what special handling they require. 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?
- Once you have a scoping plan, implement it! Our provided `Hashtable` class may come in handy for quick mapping of names to declaration.
- Reread 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++. However, this doesn't mean the variable declarations and statements within a block can be freely mixed: Doing that would be a syntax error!)
- Note there are two separate scopes for a function declaration: one for the parameters and one for the body. Note that every block of statements has its own scope, though some scopes will not introduce new bindings.
- A class's scope contains all of its fields (functions and variables). A subclass inherits all fields of its parent class. There are various ways to handle inheritance (linking the subclass scope to the parent, copying over the fields, etc.). Consider the tradeoffs and choose the strategy you will implement. Interfaces can be handled in a similar way.
- 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.

Moving on to the full implementation of the semantic analyzer:

- Start by making sure you are completely familiar with the semantic rules of Decaf as given in the specification handout. Look at the samples and examine what are the errors are in the bad files and why the good files are well-behaved.
- One 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 Booleans. Checking a `BreakStmt` would make sure it is in the context of a loop body.
- 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). You can unify these cases and have less code to write.
- The field access node is one of the trickier parts. Make sure that access to instance variables is properly enforced for the various scopes and that the implicit `this` is handled properly. Dealing with the `length()` accessor for arrays will require its own special handling, as will the special `GetByte()` and `setByte()` library methods which may operate on *any* expression.
- 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.

Matching Our Output

- Use our provided error-reporting support to match 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 should be reported from left to right.
- The node locations are used when calling the error-reporting functions to show the user where the error lies. Depending on how you assigned locations, the underlined token may vary. You do not have to match our output exactly in terms of location, but they should be in the same neighborhood.

Testing

Testing works in much the same way as PP1 and PP2. After you have acquired PP3's updated files, you should notice the `tests` directory has a new suite of tests: `samples-pp3`. Go to your `tests` directory and type "make up" to get the latest shared tests too. As before, type "make test" from your root project folder to run the tests.

Extension Ideas

As always, there are lots of ways in which you could extend the project. Here are a few ideas to get you started.

- A really big project would be to use your Decaf-To-C++ compiler and write PP-3/4 in Decaf. This way, you would be able to bootstrap your own Decaf compiler. This would be way cool.
- Grammar additions could be added with semantic analysis: the `switch` statement, postfix increment expressions, the C ternary operator, or a `foreach` construct for arrays. Comma-separated lists of variables in a declaration?
- A nice `gcc` feature is that it only reports one error per use of undeclared identifier in a scope, and suppresses errors from subsequent use (even if the later usage wasn't type-compatible with the earlier one). Even better might be inferring the type of the undeclared identifier and then concocting a declaration that allows proper type checking to continue from there.
- The Decaf spec says we don't detect a function that is declared to return a value but fails to do so before falling off the textual end of the body. But it's possible to add a compile-time check to determine this. Similarly a check for determining if variables are used before they are initialized would be welcome. Both of these require a bit of effort in control flow analysis.
- What would it take to support function overloading—the re-use of the same name for two or more functions different type signatures? How would this change how you manage identifiers and scopes? How is a call to an overloaded method resolved?
- Previous CS143 students had to implement access modifiers on each class field individually. You could support `public`, `private`, and `protected`, a la Java.
- What about support for type qualifiers such as `const/final` and `static`? How are these handled in C++ or Java? What operations should be disallowed on a `const` variable? How are `static` variables and methods in a class accessed?

Grading

Most of the grade will be allocated for correctness (85%). There will also be some consideration (15%) for design and implementation (see the "Coding Guidelines" page on the website). We will run your program with the given tests as well as others of our own. We will use `diff -w` to compare your output to the reference solutions. Credit for optional extensions is will be tracked separately and added as part of the final course grade determination.