Programming Project 3: Semantic Analysis

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 is necessary) and validate that the semantic rules of the language are being respected, printing appropriate error messages for violations. What kind of rules are being verified? Lots of type checking, for one. 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 aren't violated, 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.

Scope Checkpoint: Monday, July 30th at 11:59 p.m. (Hard Deadline)
Everything: Monday, August 6th at 11:59 p.m.

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 have plotted a "checkpoint" along the way. At the checkpoint, your analyzer needs to show it can handle errors related to scoping and declarations, because these form the foundation for the later work. We'll try to identify any issues and/or bad design decisions right away so you don't find yourself trapped after a huge time investment.

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 for your checkpoint 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 9
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 **`yylloc`** 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: g++ error messages, for instance. Oy!) 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 predefined error messages are listed in **`errors.h`** and described below. Your output is expected to match our wording and punctuation. If you have a more clever or helpful message, you are welcome to print another line after the stock error message providing more descriptive or detailed information, but the first message should be one of our predefined ones.

Checkpoint Error Messages

For the checkpoint, 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 are required to catch at the checkpoint 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. Where you find errors, print informative messages to alert the programmer to the root cause. 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 **string**s or when assigning **null** to a **double** variable. This will help 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

The starting files for this project are in the **`/usr/class/cs143/assignments/pp3`** directory. The starting project contains the following files (the boldface entries are the ones you will definitely modify, depending on your strategy you may modify others as well):

`Makefile`	builds project
`main.cc`	**`main`** and some helper functions
`scanner.h/l`	our scanner interface/implementation
`parser.y`	**`yacc`** parser for Decaf (replace with your PP2 parser)
`ast.h/.cc`	interface/implementation of base AST node class
`ast_type.h/.cc`	interface/implementation of AST type classes
`ast_decl.h/.cc`	interface/implementation of AST declaration classes
`ast_expr.h/.cc`	interface/implementation of AST expression classes
`ast_stmt.h/.cc`	interface/implementation of AST statement classes
`errors.h/.cc`	error-reporting class for you to use
`hashtable.h/.cc`	simple **`Hashtable`** template class
`list.h`	simple list template class
`location.h`	utilities for handling locations, **`yylloc`**/**`yyltype`**
`utility.h/.cc`	interface/implementation of our provided utility functions
`samples/`	directory of test input files

Copy the entire directory to your home directory. Use **`make`** to build the project. The provided **`Makefile`** will build a compiler called **`dcc`**. It reads input from **`stdin`**, writes output to **`stdout`**, and writes errors to **`stderr`**. You can use standard UNIX file redirection to read from a file and/or save the output to a file:

```
% dcc < samples/program.decaf >& program.out
```

We provide starter code that will compile but is very incomplete. It will not run properly if given sample files as input, so don't bother trying to run it against these files as given. As always, the first thing to do is to carefully read all the files we give you and make sure you understand the lay of the land. There is not much different than what you started with for PP2, so it should be fairly quick to hit the ground running. A few notes on the starter files:

- You are given the same parse tree node classes as you started with in PP2. We removed the tree printing routines to avoid clutter.
- Replace **`parser.y`** with your PP2 parser. You should not need to make changes to the parser or rearrange the grammar, but you can if you like.
- 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 templatized 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

Implementation Hints

For the checkpoint, 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 for the milestone:

- 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 and you're done with the checkpoint. Congrats!

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 through the sample files and examine what are the errors are in the bad files and why the good files are well-behaved. If you have any questions about the specification of Decaf, be sure to ask!
- 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 and debug.
- 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.
- 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.

Matching Our Output

- Please be sure to take advantage of our provided error-reporting support to make it easy for you 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 are usually reported from left to right. It's ok if your errors are a bit rearranged but follow these rules of thumb where possible.
- 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. Consider the frazzled Decaf programmer trying to fix an error and needing your help to identify where the problem is. Don't make it worse by leading them astray!

Notes on C++

- Search the web if you need to fortify your C++ skills. Our projects stay within a small subset of the language, but you are free to use as little or as much C++ as you are comfortable with. I highly recommend the use of inheritance and polymorphism in handling the similar behaviors for node classes. With a little planning, it works out beautifully.
- You will need to downcast some of your node pointers. The ordinary unsafe static C typecast can be used for this purpose, but C++ also has a safe runtime template cast. The syntax is:

```
Super *d = ….
Sub *fn = dynamic_cast<Sub *>(d);
```

  This will assign **`fn`** to the downcast **`Sub *`** if successful or **`NULL`** if **`d`** is not a **`Sub`**. It is similar to **`instanceof`** followed by a downcast in Java.
- Do not feel obligated to worry about memory leaks, even though a real compiler would need to (well, to be honest, even some production compilers leak like a sieve).

Testing

Check out the sample subdirectory for a slew of Decaf programs to chew on (some are semantically valid, and others are not.) If the source program is correct, there is no 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.

You will no doubt have questions like: "Which is the best error message for this situation?" and "Is this considered a cascade from this earlier error or not?" These are perfectly valid questions, but don't be surprised if we're a bit hesitant to answer. We want you to think through and make these sorts of decisions for yourself. Put yourself in the position of someone using your compiler, and provide the most helpful error message you can. We have the broad criteria that each error should be reported by one message from our list, but if you want to provide additional information, you can print an additional line with further details. Once an error is reported, you should suppress any subsequent errors cascading from that problem. (The project is focused on semantic errors only. We will not test on syntactically invalid input.)

The `README`

Your **`README`** file is expected to briefly explain your design decisions. Sketch how your code is structured, and explain why you believe the design is a good one (i.e. why it leads to a correct and robust semantic analyzer). You should also document the decisions you made regarding semantic error handling and anything we should know about your submission in order to properly test it.

Extra Credit

Thought of a semantic error not listed in the assignment description? Have a really nifty test case? If so, please feel free to add your own extensions to the base assignment. We'd love to see what you've come up with, and will be happy to award extra credit. That said, please make sure not to break any of the existing functionality with your new features. Any legal Decaf code should pass your semantic checker.

Grading

The checkpoint is worth 4% of your overall grade. We will use this milestone to check your progress, by running your submission through a set of tests, diff-ing your output against our solution. We will report any discrepancies back to you as feedback so you can address these issues before the final submission. The checkpoint deadline is **firm**—in order to get quick turnaround, we cannot accept any late submissions for the checkpoint. We will grade the checkpoint on a strictly pass/marginal pass/fail basis: a submission that works correctly on all or most of our tests will receive full credit, one that succeeds on at least half our tests will receive half credit, otherwise it is a zero. Not hitting the checkpoint is a sad (and totally avoidable) situation—you will have to do that work for the final submission anyway, and you will be bummed if you've lost your chance to earn free points. Don't let this happen to you!

The final PP3 submission is worth 16% of your overall grade (which, combined with the checkpoint, makes this assignment worth 20% of your overall grade). Most of the points are allocated for correctness, but we will also evaluate your design decisions (as described in your **`README`**) and the cleanliness and efficacy of your implementation. We will **`diff`** your output against our solution as a first pass, expecting that we will likely have made similar choices for most situations, but we will also be examining your output manually. The chief criteria we will use is whether your output makes sense from the point of view of a programmer using your compiler.

Deliverables

Please submit your checkpoint submission as **`pp3check`** and your final one as **`pp3`**. As always, please feel free to ask questions after lecture, during office hours, on Piazza, or over email.

Good Luck!