P3: Semantic Analysis

Due at Midnight 3/06/2023

Goal

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

Our semantic analyzer will traverse the parse tree 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.

Your finished submission will have a working semantic analyzer that reports all varieties of errors. Semantic analysis is a much bigger task than the scanner or parser. Past students speak of the semantic analyzer project in hushed and reverent tones. It is a big step up from P1 and P2. Although an expert procrastinator can start P1 or P2 the night before and still crank it out, a one-night shot at P3 is not recommended. Give yourself plenty of time to work through the issues, and thoroughly test your project.

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.

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.**

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 P1, each token's location was recorded into yylloc by the scanner. yacc then tracked the location of each symbol on the parse stack using @1, @2, etc. As part of P2, you were storing locations with the parse tree nodes, at the time you may not have realized the eventual purpose was for error-reporting. 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 compilers...) 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 **exactly**.

Here is the list of error messages from the full semantic analyzer. The portions in boldface are placeholders.

1) *Conflicting declarations*:

```
*** 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 for a mismatch when a class overrides an inherited method.

2) *Undeclared identifiers*:

*** No declaration found for **class 'Cow'**
*** No declaration found for **function 'Binky'**
*** No declaration found for **type 'Unknown'**

These messages are used to report undeclared identifiers. The error messages are also used for undeclared classes, interfaces, functions, variables. The last one is used when the identifier is supposed to be either a class or an interface.

3) *Incomplete implementations*:

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

This message is used for a class that claims to implement an interface but fails to implement one or more of the required methods.

4) *Invalid self-references*:

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

This is used to report use of this outside a class scope.

5) *Invalid use of arrays*:

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

These are used to report incorrectly formed array subscript expressions or improper use of the NewArray built-in.

6) *Incompatible operands*:

*** Incompatible operands: **double * string**
*** Incompatible operand: **! int[]**

These are 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.

7) *Invalid function invocations*:

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

These are 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.

8) *Invalid field accesses*:

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

These messages are for reporting problems with the dot operator. Field means either variable or method. The first message reports 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.

9) *Improper statements*:

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

These are for reporting improper statements. The first one is used when the test expression in a loop or conditional statement returns a non-boolean value.

We have deliberately tried to provide catch-all 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 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 cannot 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 reported again.

It will come down to judgment calls on the fringe cases. To help you making the decision about whether an error should be reported or treated as a cascading one, we have provided a sample `dcc` executable under the `solution/` directory. Try to make your own Decaf code that you'll probably have problems with and check which errors are output by the sample solution.

Starter Files

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 it with your P2 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
solution/	directory of the solution executable

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.outanderr
```

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 from what you started with for P2, 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 P2. We removed the tree-printing routines to avoid clutter.
- **Replace parser.y with your P2 parser** and call program->Check() instead of program->Print() in the first rule. You do not need to make changes to the parser or rearrange the grammar, but you can if you like. You can remove the extensions of switch statements and postfix expressions introduced in P2 since they are not part of Decaf and we won't test these constructs in this project, but it is also fine for you to keep them. You can also use the sample code of P2 once we release it.
- We've added another generic collection class for your use -- the Hashtable. It maps string keys to values. Like the List class you've already used, it is a template class to allow you to use with all different types of values. This may come in handy for managing symbol tables and scoping information. Read hashtable.h to learn more about its facilities.

Semantic Analyzer Implementation

Here is a quick sketch of the tasks that need to be performed:

- 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?

• Once you have a scoping plan, implement it. Our provided Hashtable class may come in handy for quick mapping of name to declaration.

• **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.

• A class has a scope containing 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 somewhat similarly to classes.

• 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 halfway through. Congratulations!

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 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 if both operands are of boolean type. 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. Read the spec again 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.

• 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`. Dealing with the `length()` method 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 Sample Output

• Please be sure to take advantage of our provided error-reporting support to make it easy for you to match the sample 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, so try to match your output to that of the sample solution. If you have any question on the output of the sample solution, please contact the GSI for help. Also, please check the forum frequently we may announce clarifications to those fringe cases.

Testing

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, 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.

Note: the project is focused on semantic errors only. We will not test on syntactically invalid input.

Grading

The final P3 submission is worth 100 points. We will run your submission through a complete set of tests and diff your output against our solution, expecting that your output matches the sample solution's one.

Submission

Submit your *.c, *.cc, *.cpp, *.h, *.hpp, parser.y, scanner.l to our autograder.