PP3: Semantic analysis

Due 11:59pm Fri Nov 17th
(can be submitted at most **2** days late)

The goal

In the third and fourth programming projects, your job is to implement a semantic analyzer for your compiler. This is the penultimate phase of the front-end, after this, we know the source program is free from compile-time errors and we can generate code for it. The parser will drive the process using syntax-directed translation. As each syntactically-valid construct is recognized, the parser calls additional routines to verify that the semantic rules of the language are being respected. When the rules are violated, your compiler will print out appropriate error messages. The semantic analyzer is a bigger job than the scanner or parser, so you will implement it over the next two programming projects. In pp3, you will verify the semantic validity for declarations (variables, functions, and classes) and manage scoping issues and the symbol table. In pp4, you will go on to add type-checking for all of the various expressions, field access, function calls, and so on.

Past students speak of this project in hushed and reverent tones. Although an expert procrastinator can start pp1 or pp2 the night before and still crank it out, a one-night shot at pp3 is not recommended. Our advice: get started soon. Today. Even experienced programmers usually need several days to work through all the intricacies of the assignment (though for comparison, pp3 should require less time than the larger assignments in CS106 or 107). Give yourself plenty of time to work through the issues, ask questions, and thoroughly test your project before you submit. If you run up against the wall, you can use up to two of your late days to buy extra time (you did save them, right?), but by getting it done on-time, you get the weekend to relax and recharge, with no CS143 work hanging over your head.

Semantic rules of SOOP

Since you are about to embark upon a journey to write a semantic analyzer, it helps if you first know the rules you need to enforce! SOOP is a (mostly) strongly-typed language: a specific type is associated with each variable, and the variable may contain only values belonging to that type's range of values. What are some of the other semantic rules of the language? (not all of these are relevant to pp3)

general/scopes:
- all variables, functions, and classes must be declared to prior to usage; usage of an undeclared identifier is an error
- identifiers within a scope must be unique (i.e. cannot have two local variables of same name, a global variable and function of the same name, a global variable and a class of the same name, etc.)
- identifiers re-declared or re-defined with a nested scope shadow the version in the outer scope (i.e. it is legal to have a local variable with the same name as a global variable, a function within a class can have the same name as a global function, and so on.)
- declarations in closed scopes are inaccessible
- declarations in the global scope are accessible anywhere in the program (can be shadowed though)

base types:
- the base variable types are int, double, string, and bool

constants:
- the constants are true, false, NULL, integers, doubles, string literals

arrays:	• the base type of an array can be any base type, another array or a class • the index into an array must be an int
variables:	• variables can be of base type, array type, or class type; void type is not allowed
classes:	• all classes are global, i.e., classes may not be declared inside a function • all classes must have unique names • class types must be declared or defined before use • multiple declarations of a class are allowed but it can only be defined once • if specified, the parent of a class must be a declared class type • instance variables can be of base type, array type, or class type; void type is not allowed • a field name can be used at most once within a class (i.e. cannot have two methods of the same name or a variable and method of the same name) • private members are visible only to members of the same class • a derived class can override an inherited method (replace with redefinition) but the inherited must match the original in return type and parameter types • no overloading: a class can have exactly one method with a particular name and it must have the same type signature as an inherited method of that name. • a subclass is type-compatible but not type-equivalent with its parent class (you can substitute an expression of subclass type where parent type is expected) • the use of “this.” is optional when accessing fields within a method
functions:	• functions are either global or declared with a class scope; functions may not be nested within other functions • all functions must be declared before use • function parameters and return values can be of base type, array type, or class type, parameters cannot be of void type but return type can • names cannot appear more than once in a parameter list • parameter declarations are recorded in a separate scope from the function’s local variables • a function declaration and its definition must match in type signature (type signature means return type as well as number and types of parameters, the names assigned to the parameters are not compared) • multiple declarations of a function are allowed (as long as they match), but a function can only be defined once • the number and types of arguments in a function call must match the parameters in the function definition • a return statement must return a value of the function’s declared return type
expressions:	• the two operands to any binary arithmetic or relational operator must either both be int or both be double • the operand to an unary minus must be int or double • the operands to any logical operator must be bool type
assignment:	• the left-hand side must be a variable • the types of both sides must be the same • NULL can only be assigned to a variable of class type
if/while/for:	• the test expression must be of bool type
Print:	• the argument to a print statement can only be of type string or int
New:	• the argument to New must be a declared class name
NewArray:	• the argument to NewArray must be a variable of array type

Error messages

You will not be handling all of the semantic rules above as part of pp3. For this project, you are dealing only with declarations and definitions. Here is the exact list of those error messages we expect your compiler to be able to output (at the appropriate time, of course) along with some suggestions of how to handle the error situations:

```
*** Sorry, no forward function declarations in pp3
*** Sorry, no forward class declarations in pp3
```
Just discard any forward declarations for pp3

```
*** Cannot define function 'XXX' twice
*** Cannot define class 'XXX' twice
*** Cannot use identifier 'XXX' twice in this scope
*** Cannot use identifier 'XXX' twice in parameter list
```
Keep the first definition/declaration and discard any subsequent ones

```
*** 'XXX' is not a valid class name
```
Proceed as though the variable/parameter/field was of int type

```
*** Cannot inherit from undefined class 'XXX'
```
Proceed as if the class had no parent

```
*** Cannot declare 'XXX' as void type
```
Proceed as though the variable/parameter/field was actually of int type

```
*** Overridden function 'XXX' must match inherited type signature
```
Discard the function re-definition

We would like your error messages to match ours as given above. No other error messages should be output by your compiler.

Starter files

The starting files for this project are in the **/usr/class/cs143/assignments/pp3** directory. You can access them directly on the leland filesystem or from the class Web site **http://www.stanford.edu/class/cs143/**.

The starting project contains the following files (the boldface entries are those you will modify):

`Makefile`	builds project
`main.cc`	`main()` and some helper functions
`soop.h`	defines prototypes for scanner/parser functions
`soop.l`	SOOP scanner
`soop.y`	Yacc parser for SOOP grammar, accepts soop.ebnf
`declaration.h/.cc`	interface/implementation of Declaration class
`decllist.h`	interface/implementation of our provided decl list
`hashtable.h/.cc`	interface/implementation of our provided hashtable
`scopestack.h/.cc`	interface/implementation of ScopeStack class
`semantic.h/.cc`	interface/implementation for semantic routines
`type.h/.cc`	interface/implementation of Type class
`utility.h/.cc`	interface/implementation of our provided utility functions
`samples/`	directory of test input files

Copy the entire directory to your home directory. Run `make depend` to set up dependency information and then use `make` to build the project. The makefile provided will produce a parser called `pp3`. It reads input from stdin and you can use standard UNIX file redirection to read from a file and/or save the output to a file:

```
% pp3 < samples/program.soop >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.

- The `soop.l` and `soop.y` files are our lex and yacc files for the SOOP scanner and parser. Take a look at our scanner and parser so you know we have done. You can replace our scanner and parser with your own if you prefer, but take care to ensure that you're not carrying bugs forward that will get in the way of being able to implement the additional features. You will not need to modify our `soop.l`, but `soop.y` needs some attention. It has the token types, precedence, and production rules in place, but you'll need to add actions to the rules to perform semantic processing.
- The `hashtable` and `decllist` files provide some project support that you'll definitely want to take advantage of. A Hashtable maps identifier names to Declaration objects just like the one you wrote for pp1 (useful for managing symbol tables) and a DeclList manages a list of declarations (useful for processing parameter lists). Read the .h files to learn the interface for these two objects.
- The `scopestack` files define a class skeleton for the ScopeStack class responsible for managing the stack of scopes. You will need to complete the implementation as sketched as well as define additional operations that you find necessary to support scoping. This class is very straightforward to design and implement.
- The `Declaration` and `Type` files define classes for storing Declarations and Types, respectively. A Type object represents a particular type in the program (be it a base type, an array, a class, or a function) and defines operations to compare types for equivalence. A Declaration object is little more than an identifier name associated with a Type. Both classes are fairly primitive as given and you will need to enhance them both. Start by examining what is given, so you know what you have to work with. It's probably easiest to add things on an "as-needed" basis since it may not be immediately apparent what fields and operations you will eventually want for those objects.
- The `semantic` files are provides so you can separate the semantic processing routines into their own module, rather than wedging them into the yacc file. As you discover a need for helper routines (e.g. a function that determines if two decl lists match), add it to the semantic.cc file and export it in the .h so you can call it from the parser.

Semantic analyzer implementation

You are to implement scopes and a scope stack, represent and examine type information, create and store declarations, and report a small number of error messages having to do with incorrect declarations. Mostly what you are doing is writing C code that is called when reducing various productions during parsing. For example, for a variable declaration, you call a routine to create a new declaration associating the identifier with its type and add the declaration to the symbol table, checking to be sure that it doesn't conflict. Below we outline a roadmap of our recommended plan of attack. Adopting our strategy should help you get through the tasks preserving maximum sanity.

1) Implement the scope stack.
   - Implement the ScopeStack, representing each scope as a hashtable. You can use our provided hashtable or the one you wrote for pp1.
   - Add actions to `soop.y` to push and pop scopes where appropriate. Note the ScopeStack itself is accessed as a global variable. (oh, for shame!) Print the contents of when a scope is popped to match our output.

2) Implement simple variable declarations
- Peruse the beginnings of the Type class given to you. Familiarize yourself with how the base types are stored. Implement the `Type::IsEquivalentTo()` method for base types.
- Following the example for `int` type in `soop.y`, complete the actions for the Type production for the other base types.
- Extend the Declaration class to support variable declarations.
- Add actions for the VariableDeclaration and Variable productions to create a Declaration object and declare it in the current scope.

3) Implement function definitions
- Extend the Declaration class to support function declarations, you will need to use the provided DeclList for processing/storing parameter lists.
- Add actions for the parameter and FunctionDefinition productions. There are two scopes for a function definition: one for the parameters and one for the compound statement.
- We will ignore the FunctionDeclaration production in PP3. Forward function declarations are a bit messy to deal with, and don't add any interesting issues. Our `soop.y` already has an error message for this.

4) Implement array types
- Extend the Type class to support array types. Array type equivalence should be computed using structural equality.
- Complete the actions for the ArrayModifiers production and integrate with rest of the Variable production. You may find it convenient to add a global variable here to remember the base type of the array while processing the declaration (although you can do it without a global if you're determined).

5) Implement class definitions
- Enhance the Type class to support class types. Each class will store a hash table of declarations for its fields (variables and functions). This table is the class's scope. Don't forget to add the special private variable `this` to the class's scope. Push and pop the class's scope when processing its definition.
- Now consider inheritance. A class must record its superclass (or NULL if it has no parent). A subclass should inherit all fields of its parent class. Thus, you must copy all the fields from the superclass's scope to the subclass's scope (this will require use of the iterator over the hashtable). For pp3, go ahead and copy all fields (both public and private, skipping the parent's "this" field), we will later implement access control so that the subclass can't access the private fields.
- We will ignore the ClassDeclaration production in pp3. There is an error message already in `soop.y` for this case.

6) Error messages
- When you encounter an error in a declaration, you should call our provided yyerror function to report it. All error messages have two lines, the first of which is:

```
*** Error on line 95 (last token was 'while'):
```

  Depending on where you do your error processing, the last token may vary and the line number may be off by one. You do not have to match our output exactly in terms of line number and token, but you should be in the nearby neighborhood. The line number is especially important – think of the poor, frazzled SOOP programmer trying to find where the error is in his/her program.
- Your program should not quit after it sees the first error. Upon finding a semantic error, print an error message, and then continue parsing and checking the rest of the file.

Other random hints and suggestions

- You are going to be doing some intense hacking in yacc. Be sure to make use of the documentation linked on the class web site. For this project, you will need to understand how values are passed around using the `$` notation and how the types of the non-terminals are set using `%type` definition and how that relates to `yylval`. At some point over the next three projects, you will probably need to embed actions within your productions. There is information on how to do these things (and much more) in the on-line manual, so don't forget to use it as a resource.
- Keep on your toes when assigning and using results passed from other production in yacc. If the action of a production forgets to assign `$$` or inappropriately relies on the default result (`$$=$1`), you usually don't get warnings or errors, instead you are rewarded with entertaining runtime nastiness from using an unassigned variable.
- The `PrintDebug` function in our utility module may come in handy for tracing the actions of your parser and analyzer. You can associate messages with various keys and turn the keys on and off with command-line flags which is convenient during debugging. You do not need to remove these calls in your submission, just be sure that all flags are turned off so no debugging messages print.
- You will probably need a global variable or two, but think carefully before adding another one until you are sure it is truly necessary.
- We don't recommend putting the gory processing goop directly in your actions for your productions. The preferred good yacc style is to keep the actions small by putting most of the code in separate functions in the appropriate files, then calling these functions from the rules. Besides simplifying debugging, this keeps the grammar easy to read.
- Do not free anything, even though a real compiler would need to (well, to be honest, even some production compilers leak like a sieve). This project is hard enough without subtle storage allocation bugs.

Testing your semantic analyzer

There are several test files, both correct and incorrect, and various output files, that are provided for your testing pleasure in the samples directory. The output files can be useful in understanding how the errors are reported and what happens from there. We will be testing on more files than what we have provided in the samples directory, so don't be caught unaware — consider it part of your job to think about the possibilities and devise your own test cases to ferret out any lurking issues not addressed by the set of samples.

The project focused on detecting and reporting semantic errors. The parser we give you will reject and halt on syntactically invalid programs. You do not need to add any additional processing to try to report or repair syntactically invalid input.

Grading

This project is worth 100 points, and all 100 points are for correctness. We will test the programs by diff –w the output against our solution, and examine discrepancies by hand to make sure any differences were reasonable. Even though you may be able to think of a more clever way to report an error than we suggest, please do use our wording to facilitate our verification that your compiler is working properly.

Deliverables

You may want to refer back to the pp1 handout for our general requirements about the submitted projects, electronic submissions, late days, partners, and the like. All the same rules apply here.