PP4: Semantic analysis II

Due 11:59pm Wed Nov 29th
(**no** late assignments accepted)

The goal

In the fourth programming project, your job is to finish the semantic analyzer that you started in the last project. In pp3, you laid the groundwork by building declaration and type objects to track identifiers with their associated type, class definition, or function prototype. Now you can go on to use that information to verify that the semantic rules of the language are being respected. What kind of checks does your compiler need? Lots of type-checking, for one. Only numbers can be added. The actual parameters in a function call must be compatible with the formal parameters in the function definition. Variables must be declared before use and can only be used in ways that are acceptable for the declared type. The test used in an if or while statement needs to be of boolean type. In addition to type-checking, there are also other rules to verify: access control on class fields, break statements only appear in while loops, and so on.

Unlike the earlier sequence of pp's 1, 2 and 3 where each project brings very different tasks and tools, pp4 is largely just an extension of the work you did in pp3. Once you have conquered pp3 and feel comfortable using the attribute features of yacc, you should find this project right up your alley. There are lots of details to manage as part of semantic analysis, but no fundamentally new coding techniques or tools are needed. One of the more interesting and worthwhile parts of the assignment is stepping back and thinking through what is the best way to report various errors, so as to most help the programmer fix the mistake and move on. At this point, you should start to feel like you have built most of the front-end of the compiler (well, because, you have!)

Although we are giving you the same amount of time for pp4 as we did for pp3, don't take that as an indication it is of the same magnitude, there are a few extra days for Thanksgiving. Most students find this project less complex and time-consuming than pp3, but more so than pp1 and 2. Because the starter files for pp5 must be posted right after pp4 comes in, we will not able to accept any late projects, so be prepared to submit what you have by Wednesday night without exception.

Semantic rules of SOOP

Your pp4 is responsible for reporting an error when any of the SOOP semantic rules is violated. The semantic rules are given in the pp3 handout and herea are a few additions and clarifications:

identifiers:	• it is illegal to declare an identifier named "this" (you do not have to verify or report this error, we will not test on it and you can design your code assuming it was being enforced)
arithmetic:	• the two operands to binary arithmetic operators (+,-,*, and /) must either be both int or both double • the two operands to % must both be int
relational:	• the two operands to binary relational operators (<, >, <=, >=) must either both be int or both double • the two operands to binary equality operators (!=, ==) must either be both int, both double, both boolean, or both objects (variables of any class type or NULL)
arrays:	• subscripting (e.g. [4]) can only be applied to expressions of array type

classes:

- subclasses cannot override inherited variables (whether public or private). Overridden methods must exactly match the type signature of the inherited version (whether public or private). The access level cannot be changed on an inherited method (you do not have to check or report this last one, but can assume it is being enforced)
- class fields (variables and methods) must be declared before use (i.e. unlike Java where you can refer to fields that won't be declared until later in the class definition, SOOP requires those fields to be declared before used within the class definition)
- field access (using the .) can only be applied to an expression of class type
- private fields can only be accessed in the class itself (not in subclasses or outside class), public fields are accessible anywhere
- an instance of subclass type is compatible with its parent type, and can be substituted for an expression of a parent type (e.g. if a variable is declared of type class Animal , you can assign it from a right-hand side expression of type class Cow if Cow is a subclass of Animal. Similarly, if the Binky function takes a parameter of type class Animal, it is acceptable to pass a variable of type class Cow)
- the rule above applies across multiple levels of inheritance as well
- it is the compile-time declared type of an object that determines its class for checking for fields (i.e. once you have upcast a Cow to an Animal variable, you cannot access Cow-specific additions from that variable)

statements:

- a break statement can only appear within a while loop

Error messages

The following list gives the exact list of error messages we expect you to implement. This list does not include the messages you already generated in pp3, which are assumed. The first line of each error has the form:

```
*** Error on line N:
```

It is worthwhile to take the effort to make sure the line number correctly identifies the line containing the error. Pointing some poor, burnt-out SOOP programmer to the wrong line will cause them to curse you. (Note we removed the message about "last token" since it often one symbol beyond the error, it was more likely to confuse the programmer than help them find the error.) The second line of the error message should be one of the following:

```
*** Invalid operands to binary '+'  (also for other arithmetic, relational & logical ops)
*** Invalid operand to unary '-'    (also for !)

*** Invalid lvalue in assignment
*** Incompatible types in assignment

*** Test expression must have boolean type  (for if and while)

*** Printing only supported for int and string  (for Print() built-in)

*** Argument to New must be a valid class name
*** Argument to NewArray must be an array

*** Cannot apply subscript to non-array type
*** Array subscript must be an integer

*** Called object 'XXX' is not a valid function
*** Incompatible type for argument 3 of function 'XXX'
*** Wrong number of arguments to function 'XXX'
*** Incompatible types in return
```

```
*** Cannot access field 'XXX' from non-class type
*** Class has no accessible field named 'XXX'

*** Undefined variable 'XXX'

*** break is only allowed inside a while loop
```

Notice that there are some "catch-all" error messages, i.e., we do not always provide separate error messages for all the errors we detect. For example, the "Invalid operands to operator" message can be reported when the % operator is applied to two doubles. Another example is "Incompatible types in assignment" being used for assigning NULL to a non-class variable. Feel free to use these error messages in this manner.

It is essential in this project (probably more so than the others) for you to read through the sample files that we have provided. This will give you a good sense of what you need to be checking for. SOOP is, by design, not a complete language with all the features of a C++, or even C. So, there are loopholes which can be discovered by reading the sample files.

Starter files

The starting files for this project are in the **/usr/class/cs143/assignments/pp4** 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 the ones you will need to 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 our provided ScopeStack class
`semantic.h/.cc`	interface/implementation for semantic routines
`type.h/.cc`	interface/implementation of Type class
`typecheck.h/.cc`	interface/implementation for type-checking routines
`typelist.h`	interface/implementation of our provided type list
`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 compiler called `pp4`. 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:

```
% pp4 < samples/program.soop >program.out
```

We provide starter code that will compile but is very incomplete. It will not run properly and may even crash 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.

- Most of our solution to pp3 is given in the starter project. You may find it advantageous to start with your own pp3 instead of ours because you are more familiar with its design. Developing skills for shepherding your own code through revisions is very worthwhile, but so is learning how to work with someone else's code. It's your call, but either way, we have the same standards for the correctness of your final submission.
- The `Type` and `Declaration` classes are pretty much complete. Depending on how you approach things, you may need to make changes to those classes, but likely you will only make minor tweaks if at all.
- The `semantic` and `typecheck` modules are provides so you can separate the semantic processing routines into their own modules.

Semantic analyzer implementation

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

- We recommend you start by making sure you are completely familiar with the semantic rules of SOOP, as specified in the pp3 handout and amended above. 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. If you have any questions about the specification of SOOP, be sure to ask!
- Familiarize yourself with the new pseudo base type errorType. As you will see soon, errorType is used to avoid cascading error types in parsing expressions, which is something you want to do. If you make errorType compatible with all other types, then once an error is found, you can make the type of the operation be errorType. Then, the error will be propagated through the whole expression without being reported more than once. There is an errorDecl in the Declaration class that is similarly used for propagating an error where a Declaration object is required.
- Add actions for the Constant and Expression productions. Create helper functions in typecheck.cc that can be called from the appropriate Expression productions to do the necessary checking, printing an error message if necessary, and returning the result type. There are a lot of different expression productions, make sure you handle all of them correctly!
- Do the checks for array and class field access. For example, arrays can only be indexed by an integer; for classes, you need to check that the field being used is actually a part of the given class. Make sure that the access level on fields (both instance variables and methods) is also properly enforced.
- Complete the SimpleStatement production. Be careful with assignments, since there are a few details to verify there.
- Complete the type checking related to function calls. This means matching actuals to formals, verifying the name is actually a function, and so on. To process an ExpressionList, you might find it handy to use our provided TypeList class..
- Complete any remaining checks as needed. You might want to scan the error messages and semantic rules again to make sure you have caught all of them.

Random hints and suggestions

Just a few details that didn't fit anywhere else:

- The hashtable we provide is now correctly case-sensitive, so identifiers that differ only in case are now properly distinguished.
- We are again using the C++ STL, which means that `make depend` will produce some warnings on the leland systems. Make (not depend) should run cleanly on the Solaris machines, but will print annoying warnings on the HPs. Either way, the built executable will run fine.

- There are two global variables declared in the starting project which can be retained, but you should not add any additional global variables.
- There are a few situations where you may need to create an simulated Declaration to pass back up as part of expression processing (for example, when accessing a subscripted array element). You can create a synthesized declaration with a name like "@temp" that can never be a variable name to avoid any confusion. Also, at times you may find it convenient to store special Declarations in the scope stack under similarly generated names as a way of tracking context information about the current scope. (This is perhaps not the most elegant way, but it is a fairly easy one).
- Keep soop.y uncluttered – define functions in the semantic and typecheck modules and just call them from your actions.

Testing your semantic analyzer

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 usually one error message per error. You are responsible for generating one of the error messages given above and no more.

You'll note that we haven't given you any output files! As you work through the project, you will no doubt have questions like: "Which is the best error message for this situation..." and "Would two error messages be appropriate here or not..." You may also remark to yourself: "The output for the bad files sure would be helpful..." and "It sure is mean of them to not tell us the right answer here". These are all perfectly valid questions and suggestions, but the point of pp4 is for you to make these sorts of decisions on your own. We are not going to grade by comparing the output to our solution. We are going to look at your output and see if it makes sense from the point of view of a programmer using your compiler.

So try and put yourself in the position of someone using your compiler, and provide the most helpful error message that you can. We do have the broad criteria of that each error should be reported by one error message from our list, but if you want to provide additional information, you can also print an additional line with further details. But please be sure the first message is one of the messages as given above. Once an error is reported, you should not report any subsequent cascading errors from that problem.

Grading

This project is worth 90 points, and all 90 points are for correctness. As mentioned above, we will grade your program by reading its output, being sure that it can successfully recognize a wide variety of correct programs, as well as provide helpful error reporting for incorrect programs.

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.

In honor of the lovely quirks with STL, today's quote: "If C++ is the only hammer you have, every problem starts to look like a thumb."