Project 4: Semantic analysis

Due: 9 April 2012

In this project we'll add a module to our program that does semantic analysis on the syntax trees that are output from the parser. This second pass serves two roles: it "decorates" the tree with information that would have been harder to know in the middle of parsing when the nodes were first created, and it detects various sorts of errors that would have been harder to detect in the midst of the parsing pass.

1 Grammar

The grammar that we will use will be the same as for Project 3a, with three mandatory additions: *type* can also expand to the reserved words `void` and `double`, `'/'` (division) needs to be added as a binary operator, and *top-level* can also expand to a *func-decl* defined as follows:

func-decl:
  type ident `(` *opt-params* `)` `;`

Explicit instructions on what to do to modify the parser to accept these can be found at the end of this handout.

You may (but don't have to) also accept `float-lit` values as valid *expr*s.

2 Types and identifiers

Any identifier will now be associated with additional information. Foremost among that info is an answer to the question "is it a variable or a function?" The answer to that question affects what other questions need to be asked. If the identifier is the name of a variable, it needs to further be associated with one of the declarable types. On the other hand, if it's the name of a function, it needs to be associated with a full signature (return type, parameter list)—this can be a pointer to an AST node with that info, but it has to be in there. A function name also needs to have a boolean flag indicating whether it has been defined yet or merely declared.

This information is "generated" in four places:

- top-level variable declarations declare global variables
- function declarations and definitions declare function names and their signatures
- function definitions declare local variable names
- inside-block variable declarations declare local variables

We'll assume for this project that local variables can only be declared at the top level of a function, not inside nested blocks. This information will need to be stored in tables (one global, and one for each function definition) for later reference in the type checker.

Most of the error finding will be in the type checker, but there are three kinds of errors we can detect here:

- The type `void` cannot be the type of a variable, only the return type of a function
- Multiple declarations:
  - For functions, they can be defined only once, and if they are also declared, the definition and all declarations must have the same signature. (They do not have to have the same parameter names.)
  - For variables, a name cannot be used more than once at the same level, although a local variable may legally shadow (hide) a global variable.
- Due to our assumption above, it is an error to declare any variables inside an inner block (such as the body of a `while` loop).

3 Types and expressions

In addition to the declarable types, we will use the `BOOL` token internally to mark expressions we know to be boolean—this will let us write some helpful warning messages. There are thus five types that an expression can have: `CHAR`, `INT`, `DOUBLE`, `BOOL`, and `VOID`. Each of those has an associated byte width (use a full byte for `BOOL`).

In addition to their type, expressions also need to be marked as being lvalues or rvalues. (Remember that an lvalue *can* be used as an rvalue, but an rvalue *cannot* serve as an lvalue.)

Widening conversions can be done implicitly—for instance, a parameter expecting an int can be given a char, and an int value can be assigned to a double variable. Though implicit in the source code, this will (or at least might) require some machine instructions to execute; during semantic analysis you will introduce a type conversion node into the tree to represent this process.

Narrowing conversions cannot be done implicitly—for instance, it is an error to use a double expression as an int parameter.

Expressions of type void cannot be converted into any other type.

Special case: Since we don't have a casting operator (and it would be a big pain to add one to the grammar), we'll permit narrowing conversions *only* in a direct assignment from one variable to another (i.e. both sides of the assignment are lvalues). That means that

```
ch = n;
```

is permitted by our semantic rule, but

```
ch = 3;
ch = m + n;
```

are both errors.

Every expression will thus have some type information associated to it, and expressions that are built up from other expressions will have to compute their type, and every grammar rule that interacts with exprs has some rules about what expr types are semantically valid.

4 Output

The output should be pretty similar to the output from the plain-old parser, since the output from this phase is a (decorated) abstract syntax tree.

- If a test program was invalid before we introduced the semantic analysis, it's still invalid, and should give the same error as before.

- If a test program was valid before but it has an error that is not a *parse* error, then the parser was printing out a big long tree, but this program should output a descriptive compiler error instead.
- If a test program is valid, then the output should be the full syntax tree, almost exactly as before but now with the decorations (type information, symbol tables) included.

Essentially, you should be able to reuse the existing cparser-main driver, and just tweak the various `print` methods of the AST objects to include the extra info. You might want to write a `sem_fail` function instead of just using `syn_fail`.

5 For Friday

Your assignment for Friday is to carefully read this handout and plan out the locations of the changes that will need to be made to the extant `cparser` code in order to implement the semantic analysis. In particular:

- Data. Draw out a class diagram of the existing AST hierarchy, and augment it with whatever additional data needs to be stored. Do we need a whole new class? What instance variables need to be added to the existing classes? How are we going to store the tables of identifier information?
- Functions and methods. The whole analysis process should be started by a call to a single method of a `c_file` node (the node representing the entire file). But other than that, what other methods need to be declared? Do they take any arguments? Are they methods of all AST nodes, or just some subgroup in the hierarchy? Are there any helper functions that would be useful?

Write these out and bring them to class on Friday. (This will count as a homework.) We'll discuss them for a little while and then the rest of that class period will be a work day to get a solid start on this.

6 Duedates, handing in

By Wednesday the 4th, you should have implemented the type tables, generate the error types from section 2 (no void variables, etc), and look up the type of an identifier (variable or function).

The full version is due on Monday the 9th.

On torvalds, type

```
handin cmsc445 proj4  parserdir/
```

to hand in your entire `parserdir` directory. Be sure to include a readme with any notes I'll need, and your Makefile if you have one.

As with previous project assignments, this was inspired by and adapted from a project I did when I took a compilers class at Brown.

Appendix A: upgrading the parser

To upgrade the parser to handle `void` and `double`, just add the token ids `VOID` and `DOUBLE` as a valid possibility anyplace `CHAR` or `INT` are (in my model code, this was just in `match_type` around line 420; in your version you may also have some places where a `peek()` is checked for valid values).

To upgrade the parser to handle `'/'`, modify `parse_expr3` to check the peek for `'/'` as well as `'*'`.

To upgrade the parser to parse function declarations, go to the end of `parse_top_level` and, before descending into `parse_compound_stmt`, check to see if the peek is a left curly or a left semicolon, and if it's a left semicolon, consume it and return a value that only declares the function without defining it. You may either define an additional AST type for a function header without a body, or simply re-use `func_def_node` with a body value of NULL to represent the function declarations.

Appendix B: notes on analysing various node types

Note that things to do with recording identifier types are generally done on the way *down* the tree (i.e. before processing children), while things to

do with computing expression types are done on the way back *up* the tree (i.e. after processing children).

This is not meant to be a perfectly exhaustive list of everything you need to worry about, but I did try to hit everything that was likely to be unclear and/or easily missed.

if-stmt-node, while-stmt-node: The condition needs to evaluate to something that can be used to branch. A void expression is an error; a non-boolean expression should compile but generate a warning.

return-stmt-node: Should contain an expression that evaluates to the an appropriate type to return from the current function.

assignment-expr: Left operand needs to be lvalue; right operand needs to be of a type that is assignable to the left (see the section on implicit/explicit conversions).

other binary-op-expr: Both operands need to be of legal type for op being performed; legal type conversions should be added into the tree as necessary. The type of the whole expression will be the wider of the two operand types.

var-decl: Look up ident in correct table (local or global) to be sure it isn't already declared; if it has, report an error. Otherwise, add it (again, to the correct table).

ident-expr-node: Look up ident to be sure it *is* declared, looking first in local and then in global. If not in either, it's an error. If it is declared, store variable's type in the expr node.

func-defn-node: Check global table to see if func name is not already defined (if so, error). It may have been declared without defining; if so, make sure the signature matches, and if not, record the type signature in the global table. Also, create new local table and fill it in with declared parameters.

func-decl-node (possibly implemented as func-defn-node with NULL body): Check global table to see if func name is already defined; if so, make sure the signature matches, and if not, record the type signature in the global table.

func-call-expr-node: First, look up the function name and make sure it's declared (if not, error). Then, verify that the types of its real arguments match (i.e. are convertible to) the types of the formal parameters. Finally, mark the function's declared return type as the type of the expression as a whole.