CMPS 104A Assignment 4: Symbol Tables and Type Checking

Due on Monday Nov 25, 23:59:59.

1. Overview

A symbol table maintains all of the identifiers in a program so that they can be looked up for reference. Every time a symbol definition is found, it is entered into the symbol table with attributes like the declared type and line number of the declaration. This makes it possible to look up reference during later stages of the compilation. Associated with each symbol in the symbol table is also a scope which determines from where a symbol may be referenced.

Another important part of a compiler is the type checking mechanism, which is used to verify that operators and functions are passed appropriate types and return results that are consistent. This is done by traversing the abstract syntax tree, using the information about references from the symbol table and reporting any type inconsistencies and violations.

SYNOPSIS

```
oc [-ly] [-@ flag...]  [-D string] program.oc
```

All of the requirements of previous projects also apply to this project. The scoping analysis and type checking will be done after the parser has returned to the main program. For any input file called *program*`.oc` generate a symbol table file called *program*`.sym` and also generate all files specified in earlier projects. In addition, do a second pass over the abstract syntax tree after the symbol tables are created and do a type checking.

2. Symbols in `oc`

Function and variable names are both entered into the same symbol table. All functions have global scope, but variables may have global or local scope, which is nested arbitrarily deeply. Variables declared in inner scopes hide identifiers of the same name in the any outer scopes. A variable may only be referenced from the point of declaration to the end

of the scope in which it is declared.

Type names consist of the reserved words `void`, `bool`, `char`, `int`, `string`, and names defined via `struct` definitions. All type names are global and exist in a name space separate from those of ordinary identifiers.

Field names are identifiers which may only be used immediately following the dot (.) operator for field selection. The available field names depend on the `struct` definition, therefore it is not necessary to have additional symbol tables for fields.

3. Symbol Tables

The symbol table for variables and functions actually contains one table for each block, as local variables hide outer global variables. In C++, each such table can be implemented with a `std::map` using the name of the identifier as key and the type as value. If the identifier cannot be found in the inner table for the current block, it will be looked up in the table of the surrounding block. To achieve this, each table except for the global table should also link to its parent table. Figure 1 shows how nested symbol tables for the variables and functions of a simple example program could look like.

```
struct node { int foo;}
string[] a = new string[10];
char f(int b) {
  node c = new node();
  if (b < 42) {
    char c = 'x';
  } else {
    int d = 10;
  }
}
```

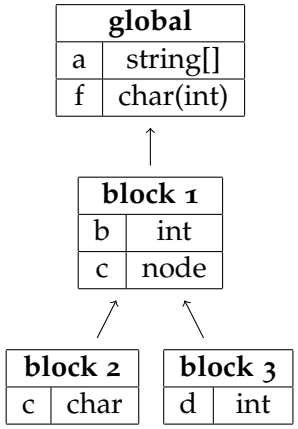

Figure 1: Symbol tables for the example `oc` program

You can either implement the nested symbol table data structure yourself, e.g. by using a hash-stack, or you can use the symbol table available at `http://www2.ucsc.edu/courses/cmps104a-wm/:/TA/asg4/` as a starting point. The starting code maps indentifiers to types; you will have to extend it to keep track of the locations of definitions.

Output to the `.sym` file should list all variable and function declarations. Each line should include the name of the identifier, the location where it is defined in `(file.line.character)` numeric format, the block number in curly braces and the type information. Global definitions should appear first. Then, all local variable declarations should be listed for each of the nested blocks in the program. For each level, the definition should be indented with 3 spaces. Listing 1 lists the expected output for the `oc` program in Figure 1.

```
a (0.2.9) {0} string[]
f (0.3.5) {0} char(int)
   b (0.3.11) {1} int
   c (0.4.7) {1} node
      c (0.6.9) {2} char
      d (0.7.8) {3} int
```

Listing 1: Expected `.sym` file output for the `oc` program in Figure 1.

4. Type Checking

The type checking relies on the information in the symbol tables and involves a post-order depth-first traversal of the AST. A detailed partial context-sensitive type checking grammar is shown in Table 1. For each violation of any of these type checking rules, print an error message to `stderr`, e.g. by using `errprintf` from `auxlib.h`, and continue the traversal to find as many type errors as possible. Your program should return a non-zero exit code if any errors occurred during the type checking.

1. Two types are compatible if they are exactly the same type. Additionally, *null* is compatible with any non-primitive type. In the type checking grammar, in each rule, types in italics must be substituted consistently by compatible types.
2. Only expressions have associated types, not statements.
3. The type of an identifier is simply looked up from the symbol table.
4. For the field selector `'.'`, the left operand must be a struct type or an error message is generated. Additionally, the field name must exists in the `'structdef'` of the left operand. The resulting type of the field selector is the type of the field as found in the `'structdef'`.
5. For a call, evaluate the types of each of the arguments, and look up the function in the identifier table. Then verify that the arguments are compatible with the parameters and error if not, or if the argument and parameter lists are not of the same length. The call node has the result type of the function.
6. The expression operand of both if and while must be of type *bool*.
7. If the function's return type is not *void*, then it must have an expression which is compatible with the declared return type. It is an error for a return statement to have an operand if the function's return type is void. A global return statement is considered to be in a *void* function.
8. The indexing operator for an array returns the address of one of its elements, and for a string, the address of one of its characters.

typedecl IDENT '=' *anytype*	→	∅
'while' '(' *bool* ')'	→	∅
'if' '(' *bool* ')'	→	∅
'return' *compatible*	→	∅
anytype '=' *anytype*	→	*anytype*
anytype '==' *anytype*	→	*bool*
anytype '!=' *anytype*	→	*bool*
primitive '<' *primitive*	→	*bool*
primitive '<=' *primitive*	→	*bool*
primitive '>' *primitive*	→	*bool*
primitive '>=' *primitive*	→	*bool*
int '+' *int*	→	*int*
int '-' *int*	→	*int*
int '*' *int*	→	*int*
int '/' *int*	→	*int*
int '%' *int*	→	*int*
'+' *int*	→	*int*
'-' *int*	→	*int*
'!' *bool*	→	*bool*
'ord' *char*	→	*int*
'chr' *int*	→	*char*
'new' *basetype* '(' ')'	→	*basetype*
'new' *basetype* '[' *int* ']'	→	*basetype []*
IDENT '(' *compatible*... ')'	→	(function lookup)
IDENT	→	(symbol lookup)
basetype [] '[' *int* ']'	→	*basetype*
string '[' *int* ']'	→	*char*
structtype '.' IDENT	→	(structdef lookup)
INTCONT	→	*int*
CHARCON	→	*char*
STRINGCON	→	*string*
'false'	→	*bool*
'true'	→	*bool*
'null'	→	*null*

Table 1: Type grammar for oc. *primitive* is either *bool*, *char* or *int*, *basetype* is either *primitive*, a *string* or *struct*, *anytype* is either *basetype* or an array of *basetype*, and *compatible* types depend on the context. ∅ denotes no type.

5. Grading

There are 60 points in total.

(5) Good overall coding style (identifiers, indentation, comments)
(2) A hierarchical symbol table with nested scopes
(2) A separate table for types (in particular `structdefs`)
(3) Symbol table keeps track of definition locations
(8) Abstract tree traversal code which builds up the symbol tables
(10) Abstract tree traversal code which checks type consistency for expressions, calls, declarations and variables.

If your program did not compile, ignore the following...

(12) Non-zero exit code for `oc` programs starting with `9x` and for `nosuch.oc`, zero exit code for all other test files (-1 pt/error)
(5) Error output for `91-typecheck.oc` and `93-semantics.oc` includes warnings for unknown variables (1), unknown types (1), and incompatible types for operations (1), calls (1) and variable declarations (1).
(3) `.str`, `.tok` and `.ast` files created as per previous project specs
(3) `.sym` files created with all variable declarations, function definitions and function parameters
(2) `.sym` file is correctly indented according to the scope nesting
(1) Entries in the `.sym` file include the unique number of the block in which they appear ({`0`} for global)
(2) Entries in the `.sym` file include the types. Arrays are indicated by a trailing `[]` and functions as their return type followed by argument types in parenthesis, e.g. `void(int)`
(2) Entries in the `.sym` file include definition locations. Locations are in `(file.line.character)` format.

6. What to Submit

Submit `README`, `Makefile`, `scanner.l`, `parser.y`, and all of the header and C++ implementation files. Do not submit the file generated by `flex` or `bison`. To submit, log into the Linux Lab, either locally or by ssh, and

```
$ cd /path/to/my/files/
Check that README (and optionally PARTNER) exists
$ /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments/grading/mk.build
```

```
Check if there are any errors building oc
$ /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments/grading/mk.tests
Check if there are any unexpected results
$ submit cmps104a-wm.f13 asg4 README Makefile ... ...
```