$Id: asg4-symbols-types.mm,v 1.14 2017-11-02 16:05:22-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments
URL: http://www2.ucsc.edu/courses/cmps104a-wm/:/Assignments/

1. Overview

A symbol table maintains all of the identifiers in a program so that they can be looked up when referenced. Every time a symbol definition is found, it is entered into the symbol table, and every time a reference is found, it is looked up in the symbol table. It also maintains scope and range information, information that 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 types of an appropriate nature and return results that are consistent. This is done by adding attributes to each node of the AST, although only declaration and expression nodes have non-null attributes.

SYNOPSIS

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

All of the requirements for all previous projects are included in this project. The semantic routines of this project will be called after the parser has returned to the main program. For any input file called *program*`.oc` generate a symbol table file called *program*`.sym`. In addition, change the AST printing function from the previous project so that it prints out all attributes in each node on each line after the information required in the previous project. Thus, the AST file for this project will have more information in it than for the previous project.

2. Symbols in `oc`

Symbols in `oc` are all identifiers, since there is no possibility of user-overloading of operators. The symbol table module in `oc` must maintain multiple symbol tables, for function and variable names, for type names, and for fields of structures. There are three classes of identifiers:

(a) **Type names:** Type names consist of the reserved words `void`, `int`, `string`, and identifier names defined via `struct` definitions. All type names are global and exist in a name space separate from those of ordinary identifiers, so type identifiers are entered into the type name table.

(b) **Field names:** Field names are identifiers which may only be used immediately following the dot (`.`) operator for field selection. The same field name may be used in different `struct` definitions without conflict, and hence are all global. Each `struct` definition causes a separate field symbol table to be created and accessed from the type name table.

(c) **Function and variable names:** All function and variable identifiers are entered into the identifier symbol tables. All functions have global scope, but variables may have global or local scope, which is nested arbitrarily deeply. A variable declared in a local scope hides an identifier with the same name in a more global scope. Variables in disjoint scopes may have the same name. A variable may only be referenced from the point of declaration to the end of the scope in which it is declared.

2.1 Categories and types in `oc`

There are three categories of types in **oc**, and each has groups of types within it. Each identifier and field has a particular type associated with it, and each **struct** defines a new type by name. Type checking of functions is done by structural equivalence but checking of variables is done via name equivalence.

(a) The void type is neither primitive nor reference.

  (i) **void**: may only be used as the return type of a function. It is an error to declare variables, parameters, or fields of this type.

(b) Primitive types are represented inline inside **struct**s and local or global variables, and whose values disappear when the block containing them leaves its scope, or the **struct** containing them becomes unreachable.

  (i) **int**: is a signed two's complement integer.

(c) Reference types are pointers to objects on the heap. They themselves may be local, global, or fields, as may primitive types, but all object types reside on the heap. Pointer arithmetic is prohibited.

  (i) **null**: has the single constant **null**. The syntax prevents it from being used in a type declaration.

  (ii) **string**: is effectivly an array of characters and has string constants associated with it. Its size is fixed when allocated. The length of the string contained in the array varies up to the maximum allocated length, depending on where the null plug (`'\0'`) is placed. Strings are stored in the same format as in C.

  (iii) **struct** *typeid*: may have as many fields as needed inside of it. It may contain both primitive and reference types.

  (iv) *base*`[]`: contains a collection of other elements, all of which are of the same type, which may be either primitive or reference types. Its base type may not be an array type. This is the only polymorphic type.

2.2 Attributes for types and properties

Each node in the AST has a set of attributes associated with it, as does each entry in the symbol tables. Attributes indicate properties associated with parts of the AST and are used for code generation. The attributes are gathered into several categories:

(a) Type attributes, all of which are mutually exclusive: **void**, **int**, **null**, **string**, and **struct**. If the **struct** attribute is present, an associated typeid must also be present.

(b) The **array** attribute which may occur with any primitive or reference type. Arrays of arrays are not permitted.

(c) Attributes describing typeids, identifiers, and fields are mutually exclusive: **function**, **variable**, **field**, and **typeid**. In addition, the **param** attribute is set if the variable is in a function's parameter list.

(d) The `lval` attribute appears on any node in the AST which can receive an assignment. This includes all variables (global, local, or parameter) and the result of the indexing (`[]`) and selector (`.`) operators.

(e) The `const` attribute is set on all constants of type `int`, `null`, and `string`. The `lval` and `const` attributes are mutually exclusive.

(f) The `vreg` attribute is set on interior nodes that hold a computed value and the `vaddr` attribute on a computed address referring to a location in memory. They are mutually exclusive. They represent virtual registers used in code generation.

Attributes can be represented as an enumeration, with `bitset_size`, the last attribute, having the largest magnitude.

```
enum { ATTR_void, ATTR_int, ATTR_null, ATTR_string,
       ATTR_struct, ATTR_array, ATTR_function, ATTR_variable,
       ATTR_field, ATTR_typeid, ATTR_param, ATTR_lval, ATTR_const,
       ATTR_vreg, ATTR_vaddr, ATTR_bitset_size,
};
using attr_bitset = bitset<ATTR_bitset_size>;
```

2.3 Polymorphism

Polymorphism is present in many languages and derives from the Greek πολυμορφισμός, meaning "having many forms".

(a) Universal polymorphism :

(i) Parametric polymorphism (universal), implemented as generics in Java, and as templates in C++, allows type parameters to be passed into data structures. In `oc`, only arrays exhibit limited parametric polymorphism, in that there can be an array of any other type except arrays.

(ii) Inclusion polymorphism (universal), known also as inheritance in object-oriented programming lanugages, allows function overriding. None in `oc`.

(b) Ad hoc polymorphism :

(i) Overloading polymorphism (ad hoc) means that the same function or operator may be defined multiple times and selected based on the types of its arguments. In `oc` operator overloading is permitted only for the assignment and comparison operators, which can have many types as arguments, provided that they are compatible, and for the indexing operator, whose left operand may be an array of any type, or a string. No functions or other operators are overloaded.

(ii) Conversion polymorphism (ad hoc) has arguments to functions implicitly converted from one type to another in order to satisfy parameter passing. None in `oc`.

2.4 Type checking

Type checking involves a post-order depth-first traversal of the AST. A detailed partial context-sensitive type checking grammar is shown in Figure 1. The following names are used : *primitive* is any primitive type, *base* is any type that can be used

as a base type for an array, and *any* is either primitive or reference.

(a) Two types are *compatible* if they are exactly the same type; or if one type is any reference type and the other is `null`. In the type checking grammar, in each rule, types in italics must be substituted consistently by compatible types. Types are compatible only if the `array` attribute is on for both or off for both.

(b) When the right side of a production is empty, there are no type attributes. Only expressions have type attributes, not statements.

(c) The result type of assignment (`=`) is the type of its left operand.

(d) Fields following a selector have the `field` attribute, but no type attribute, since their type depends on the structure from which they are selected.

(e) Identifiers have the type attributes that they derive from the symbol table. In addition, either the `function` or `variable` attribute will be present, and for variables that are parameters, also the `param` attribute. All variables also have the `lval` attribute.

(f) Field selection sets the selector (`.`) attribute as follows: The left operand must be a `struct` type or an error message is generated. Look up the field in the structure and copy its type attributes to the selector, removing the `field` attribute and adding the `vaddr` attribute.

(g) 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, with the `vreg` attribute.

(h) The expression operand of both `if` and `while` is considered false if it is a 0 or null. Any non-zero primitive or non-null pointer is considered true.

(i) 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.

(j) 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. When an `int` is stored in a `string`, only the low-order 8 bits are stored.

3. Symbol tables

Symbol tables must be maintained for identifiers, `struct`s, and for each `struct`, a field table. Figure 2 contains declarations for type `symbol` in the left column and descriptions in the right column. A symbol table is just a hash table with the identifier as the key and a symbol as the value:

```
using symbol_table = unordered_map<string*,symbol*>;
using symbol_entry = symbol_table::value_type;
```

Each entry in the symbol table is a `pair<const string*,symbol*>`. The key is a pointer into the `stringset` as returned by the intern function. This key is the same pointer as found in the AST node of the identifier definition.

3.1 The structure and field tables

All structures and fields belong to block 0 (by default), since no structure definitions may be local. Structure and field names are in separate namespaces from that of identifiers. Maintain a `symbol_table` containing all structure definitions, with the type name as the key and a field table as the value. Each structure definition points at its own field table.

3.2 The symbol-stack data structure

When handling nested symbol tables, it is necessary to efficiently create a new scope on entry, look up symbols quickly in $O(1)$ time.

(a) Maintain a vector of symbol tables, with each entry in the vector representing an open block : `vector<symbol_table*> symbol_stack;`

(b) Maintain a global counter `next_block` which is initially set to 1 and represents the next block number to be used. Each block must be unique.

(c) When entering a block, increment `next_block` and push `nullptr` onto the top of the symbol stack.

(d) When leaving a block, pop the top of the symbol stack.

(e) When defining an identifier, define a new `symbol` and insert it into the symbol table at the top of the `symbol_stack`. Create the symbol table if it is null. Not all blocks have symbols, so do not create a symbol table for a given block until it has at least one symbol defined in it.

(f) When searching for an identifier, start with the top of the symbol stack and check each non-null pointer to a symbol table, searching downward. Local identifiers hide more global ones.

4. Traversing the abstract syntax tree

Write a function that does a depth-first traversal of the abstract syntax tree. At this point, you may assume it is correctly constructed. If not, go back and fix your parser.

(a) For all nodes not involving declarations, proceed left to right in a depth-first post-order traversal, assigning attributes as necessary. All identifiers must be declared before they are used, except when a *TYPEID* declares a field of a structure, so the scan must be done from left to right, with all declarations preceding all references.

(b) Whenever a structure definition is found, create a new entry in the structure hash, using the typeid as a key. The block number is 0 because structures are declared globally. Then create a hash table and insert each field into that hash table, pointed at by this structure table entry. Field names are also in block 0.

(c) The structure name must be inserted into the structure hash before the field table is processed, because type type of a field may be the structure itself.

(d) If a structure name is found that is not in the hash, insert it with a null pointer for the field table. If it later becomes defined, fill in the fields.

(e) If an incomplete structure has a field selected from it, or if it follows `new`, or if it used in a declaration of other than a field, print an error message about referring to an incomplete type.

(f) All other identifiers are inserted into the main symbol tables.

(g) Whenever you see a block, increment the global block count and push that number on the block count stack. Then store the block number in the AST node and traverse the block. When leaving a block, pop the block number from the stack. Each block will have a unique number, with 0 being the global block, and the others numbered in sequence 1, 2, 3, etc.

(h) Whenever you see a function or prototype, perform the block entering operation, and traverse the function. Treat the function as if it were a block. The parameters are inserted into the symbol table as owned by the function's block.

(i) If the function is already in the symbol table, but only as a prototype, match the prototype of the new function with the previous one of the same name. If they differ, print an error message. If the function is already in the symbol table as a function, print an error message about a duplicate declaration.

(j) If the function is not in the symbol table, enter it, along with its parameters. If this is an actual function, traverse the block as you normally would, with the block number being the next one in line. A function creates at least two blocks, one for itself, and one for the block of statements that it owns.

(k) Whenever you see a variable declaration, look it up in the symbol table. Print a duplicate declaration error message if it is in the symbol table at the top of the symbol vector stack.

(l) If it is not found, enter it into the symbol table at the top of the symbol stack and set the attributes and other fields as appropriate.

(m) In the scanner and parser, error messages were printed using a global coördinate maintained by the scanner. In this assignment and the next, all error messages must be the coördinates in some appropriate AST node, since the global coördinate at this time indicates end of file.

5. Generated output

You must generate all output from the previous projects, and in addition, create a file with the symbol table in it with a suffix of `.sym`. In addition, additional information will be printed into the `.ast` file when traversing and printing the AST. Figure 4 shows some sample output to the `.sym` file for some sample input shown in Figure 3.

(a) Retrofit your scanner so that new fields are added to a token node when created: `attr_bitset attributes`, initialized to 0, a block number initialized to 0, and a pointer to a structure table node, initialized to null,

(b) Retrofit your parser so that its output lines look like:

```
'+' "+" (0.6.3) {4} int vreg
IDENT "foo" (0.6.8) {4} int variable (0.2.9)
IDENT "bar" (0.7.3) {5} struct "node" variable (0.3.7)
```

Following the token coordinate is the block number in which the node occurs,

and then the various attributes associated with that node. For identifiers, the last number is the coordinate of the declaration.

(c) In the traversal described above, put a block number on every node in the tree, as well as appropriate attributes. If the `struct` attribute is set, also print the name of the structure. For variables, functions, typeids, and fields, print out the coördinates of the defining occurrence of that identifier. This means that you AST file must be generated after the symbol table semantic routines traverse the AST.

(d) Output to the `.sym` file should show all identifiers, typeids, and fields listed in the same order as on input, i.e., sorted by serial number.

(e) For each definition of a variable, function, structure, or field, print out the same information into the `.sym` file. List all global definitions against the left margin and all field, parameter, and local definitions indented by three spaces. Print an empty line in front of each global definition and between the parameter list and local variables.

identdecl '`=`' *compatible* → '`return`' *compatible* → *any* `lval` '`=`' *any* → *any* `vreg`
any '`==`' *any* → `int vreg`
any '`!=`' *any* → `int vreg`
any '`<`' *any* → `int vreg`
any '`<=`' *any* → `int vreg`
any '`>`' *any* → `int vreg`
any '`>=`' *any* → `int vreg`
`int` '`+`' `int` → `int vreg`
`int` '`-`' `int` → `int vreg`
`int` '`*`' `int` → `int vreg`
`int` '`/`' `int` → `int vreg`
`int` '`%`' `int` → `int vreg`
	'`+`' `int` → `int vreg`
	'`-`' `int` → `int vreg`
	'`!`' `int` → `int vreg`
	'`new`' *TYPEID* '`(`' '`)`' → *TYPEID* `vreg`
	'`new`' '`string`' '`(`' `int` '`)`' → `string vreg`
	'`new`' *base* '`[`' `int` '`]`' → *base* `[] vreg`
	IDENT '`(`' *compatible* '`)`' → *lookup* `vreg`
	IDENT → *lookup*
	base `[]` '`[`' `int` '`]`' → *base* `vaddr lval`
	'`string`' '`[`' `int` '`]`' → `int vaddr lval`
	IDENT '`.`' *FIELD* → *lookup* `vaddr lval`
	INTCON → `int const`
	CHARCON → `int const`
	STRINGCON → `string const`
	'`null`' → `null const`

Figure 1. Type checking grammar

`struct symbol {`	Each node in the symbol table must have information associated with the identifier. It will be simpler to make nodes in all of the symbol tables identical, and null out unnecessary fields.
`attr_bitset attributes;`	Symbol attributes, as described earlier.
`symbol_table* fields;`	A pointer to the field table if this symbol is a `struct` (if the `typeid` attribute is present). Null otherwise.
`size_t filenr, linenr, offset;`	The index into the filename vector, along with the line number and offset where this symbol was declared.
`size_t block_nr;`	The block number to which this symbol belongs. Block 0 is the global block, and positive increasing sequential numbers being assigned to nested blocks.
`vector<symbol*>* parameters;`	A symbol node pointer which points at the parameter list. For a function, points at a vector of parameters. Null otherwise.

Figure 2. Symbol table node

```
struct node { int foo; node link; }
node func (node head, int length) {
   int a = 0; string b = ""; node c = new node();
   if (a < 3) { int d = 8; a = length; c = c.link; }
         else { string e = "";
                if (0 == 0) { int f = 8; }
                       else { int g = 9; }
              }
}
node h = func (null, 10);
```

Figure 3. Example program used to illustrate `.sym` file

```
node (0.1.7) {0} struct "node"
   foo (0.1.18) field {node} int
   link (0.1.27) field {node} struct "node"

func (0.2.5) {0} struct "node" function
   head (0.2.15) {1} struct "node" variable lval param
   length (0.2.24) {1} int variable lval param

   a (0.3.7) {1} int variable lval
   b (0.3.14) {1} string variable lval
   c (0.3.24) {1} struct "node" variable lval
      d (0.4.19) {2} int variable lval
      e (0.5.17) {3} string variable lval
         f (0.6.30) {4} int variable lval
         g (0.7.30) {5} int variable lval

h (0.8.5) {0} struct "node" variable lval
```

Figure 4. Sample output to `.sym` file from Figure 3