`$Id: asg4-symbols-types.mm,v 1.23 2012-11-19 14:08:24-08 - - $`
`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 reference. 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. There are three classes of identifiers :

(a) **Function and variable names :** All function and variable identifiers are entered into the same symbol table. 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 of a name in the global scope, but it is an error if a variable in an inner scope has the same name as one in an outer local scope in which it is nested. 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.

(b) **Type names :** 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.

(c) **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.

The symbol table module in `oc` must therefore maintain $n+2$ symbol tables, namely, one for identifiers, one for type names, and n for **`struct`** names, where n is the number of **`struct`**s defined in the program.

2.1 Categories of types, 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) **`void`** 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) `bool` : has constants `false` and `true`, and is an unsigned 8-bit byte.

  (ii) `char` : is an unsigned 8-bit byte. Values between 0x00 and 0x7F being ASCII characters. Values between 0x80 and 0xFF are locale dependent.

  (iii) `int` : is a signed 32-bit 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. The size of a pointer is dependent on the target architecture. 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.

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

  (iv) *basetype*`[]` : 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 generic 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`, `bool`, `char`, `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 type and with any reference type except `null`.

(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 `lvalue` 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 `bool`, `char`, `int`, `null`, and `string`.

(f) The `vreg` and `vaddr` attributes are set on interior nodes that hold a computed value or address and used in register allocation.

When representing these attributes as a bitset, use the following bit numbers, `void` (0), `bool` (1), `char` (2), `int` (3), `null` (4), `string` (5), `struct` (6), `array` (7), `function` (8), `variable` (9), `field` (10), `typeid` (11), `param` (12), `lvalue` (13), `const` (14), `vreg` (15), `vaddr` (16), where bit 0 is the least significant bit.

2.3 Polymorphism

Polymorphism is present in many languages and derives from the Greek πολυμορφισμός, meaning "having many forms". There are two types with two variants each :

(a) Universal polymorphism is either parametric or inclusion.

  (i) Parametric polymorphism, known as generic in Java, and 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, known also as inheritance in object-oriented programming lanugages, allows function overriding. None in `oc`.

(b) Ad hoc polymorphism is either overloading or conversion.

(i) Overloading polymorphism 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 has arguments to functions implicitly converted from one type to another in order to satisfy parameter passing. None in `oc`.

```
identdecl '=' compatible →
'while' '(' bool ')' →
'if' '(' bool ')' →
'return' compatible →
anytype lvalue '=' anytype → anytype vreg
anytype '==' anytype → bool vreg
anytype '!=' anytype → bool vreg
primitive '<' primitive → bool vreg
primitive '<=' primitive → bool vreg
primitive '>' primitive → bool vreg
primitive '>=' primitive → bool 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

'!' bool → bool vreg
'ord' char → int vreg
'chr' int → char vreg
'new' TYPEID '(' ')' → TYPEID vreg
'new' 'string' '(' int ')' → string vreg
'new' basetype '[' int ']' → basetype[] vreg
IDENT '(' compatible ')' → symbol.lookup
IDENT → symbol.lookup
basetype[] '[' int ']' → basetype vaddr lvalue
'string' '[' int ']' → char vaddr lvalue
TYPEID '.' FIELD → symbol.lookup vaddr lvalue
INTCON → int const
CHARCON → char const
STRINGCON → string const
'false' → bool const
'true' → bool const
'null' → null const
```

Figure 1. Type checking grammar

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, *basetype* is any type that can be used as a base type for an array, and *anytype* 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.

(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 `lvalue` 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` must be of type `bool`.

(i) In the rule statement → expr, the expression must be of type `void`, or a warning is printed indicating that a value is being ignored. Exception: if the root of the expression is the assignment operator (`=`), no warning is given.

(j) 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.

(k) 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.

3. Symbol tables

A symbol table must be maintained for identifiers, another for structures, and for each structure, a separate field table. The symbol table's general structures is a hash-stack.

Each node in the symbol table must have information associated with the identifier. It will probably be easier to make nodes in all of the symbol tables identical, and just zero or null out unnecessary fields. The following fields are required:

(a) A pointer to the string table entry, identical to that in an AST node.

(b) Attributes, represented as a `bitset_t`.

(c) A pointer to the structure table node, if the `typeid` attribute is present, otherwise null.

(d) A pointer to the field table if this is a structure table node, otherwise null.

(e) The file, line, and offset fields from the identifier AST node of the declaration.

(f) The block number to which this identifier belongs, block 0 being the global block, and positive increasing sequential numbers being assigned to nested blocks.

(g) A symbol node pointer which points at the parameter list. For a function, point at the first parameter, for other parameters except the last to the next parameter in sequence. Null if neither the `function` nor `param` attributes are present.

(h) A link to the next identifier on the symbol table stack. Nodes are popped off this stack whenever they leave scope.

(i) A collision resolution link, if the hash table uses collision resolution by separate chaining. Not necessary if linear probing is used.

The same structure, although a different hash table, can be used to store typeids defined by the `struct` declaration. And for each structure declaration, there must be a separate hash table showing all fields, and the order in which they are declared. The stack link can show this in reverse.

3.1 The hash-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, and exit a stack with no more than $O(m)$ steps, where m is the number of identifiers in the scope being closed.

(a) A data structure which satisfies this is a hash-stack, i.e., a combination of a hash table and a stack. Also, maintain a separate stack of block numbers.

(b) Initially push the global block number 0 onto the block stack and have the other two data structures empty. When entering a block, increment *nextblock* and push it onto the block stack.

(c) When defining an identifier, allocate a new node, push it onto the identifier stack, and also insert the same node into the hash table.

(d) When searching for an identifier, use the hash table.

(e) Instead of recomputing the string hash code, for searching, insertion, and deletion, the address of the node itself as a `uintptr_t` can be used, remainder the dimension of the hash table. Include `<inttypes.h>` for this type. See `misc-code/uintptr_t.c` example.

(f) When leaving a block, repeatedly pop the identifier stack of all nodes with the same block number as the top of the block stack, and remove these identifiers from the hash table. Then pop the block number off the block stack. The symbol node can not be freed at this point because it is still referenced by an AST node.

(g) Always make sure the hash table's loading factor is 0.5 or less for collision resolution by chaining and 0.3 or less for collision resolution by linear probing. It should have a dimension of $2^k - 1$ for some integer k. Perform array doubling if the loading factor exceeds the limit. Given some integer n, one can round it up the the next value of $2^k - 1$ by the expression
```
n = (1 << (int) ceil (log2 (n))) - 1
```

(h) If using collision resolution by chaining, it is necessary for new entries into the hash table to be inserted at the front of a hash chain.

(i) If using collision resolution by linear probing, the search must continue to find the last identifier with a given key. Also, removing an identifier means that its pointer must be replaced by `EMPTY`, not `NULL`. After replacing it by `EMPTY`, search forward to either an entry or a `NULL`. If a null is found, move backwards, replacing empties by nulls until either an entry or a null is found.

(j) The identifier stack can be maintained simply by keeping a single pointer to the top of the stack and then threading the nodes themselves through the stack (putting stack links into the identifier nodes).

(k) The structure and field symbol tables can use the same kind of data structure, but all block numbers are 0 and there is no removal needed. However, the same data structure can be used.

3.2 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 table.

(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 the current top of the block stack is block 0 and the identifier is already there, or if it is already in the symbol table in a nested block. This prevents an inner nested block declaration from hiding an outer nested block declaration.

(l) If it is not found, enter it into the symbol table and push it onto the symbol stack, setting up the attributes and other fields as appropriate.

(m) When leaving a scope (block), pop the block off of the block stack, and also pop all identifiers off the identifier stack if they have the same block number that you are leaving, and remove them from the hash table.

(n) The identifier symbol table is the only table from which entries are removed. Note that you can not free the nodes, since they will be used at code generation time.

(o) 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.

4. 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.

(a) Retrofit your scanner so that new fields are added to a token node when created: **`bitset_t 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)
```

(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.

```
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 2. Example program used to illustrate `.sym` file

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

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

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

h (0.8.5) {0} struct "node" function
```

Figure 3. Sample output to `.sym` file from program in Figure 2

`1 << n`	$\{n\}$
`S | T`	$S \cup T$
`S & T`	$S \cap T$
`S & (1 << n)`	$x \mid \text{ if } n \in S \text{ then } x \neq 0 \text{ else } x = 0$
`S |= 1 << n`	$S \leftarrow S \cup \{n\}$
`~ S`	$\overline{S}$
`S = S & ~ (1 << n)`	$S \leftarrow S - \{n\}$

Figure 4. Attributes in C and set theory

5. Tutorial on bitsets

Sets can be represented in C as bitsets by defining a bitset as :

```
#include <inttypes.h>
typedef uint64_t bitset_t;
```

This will allow each bit in the word to indicate whether or not the item is in the set. The expression

```
CHAR_BIT * sizeof (bitset_t)
```

can be used to determine how many bits are permitted in a bitset. The macro `CHAR_BIT` is found in `<limits.h>`. Set operations can be represented as shown in Figure 4 To see how to code and translate attributes to print, see the `misc-code/attributes.c` example.