Lecture #22

Symbol Tables and Graphical Representations

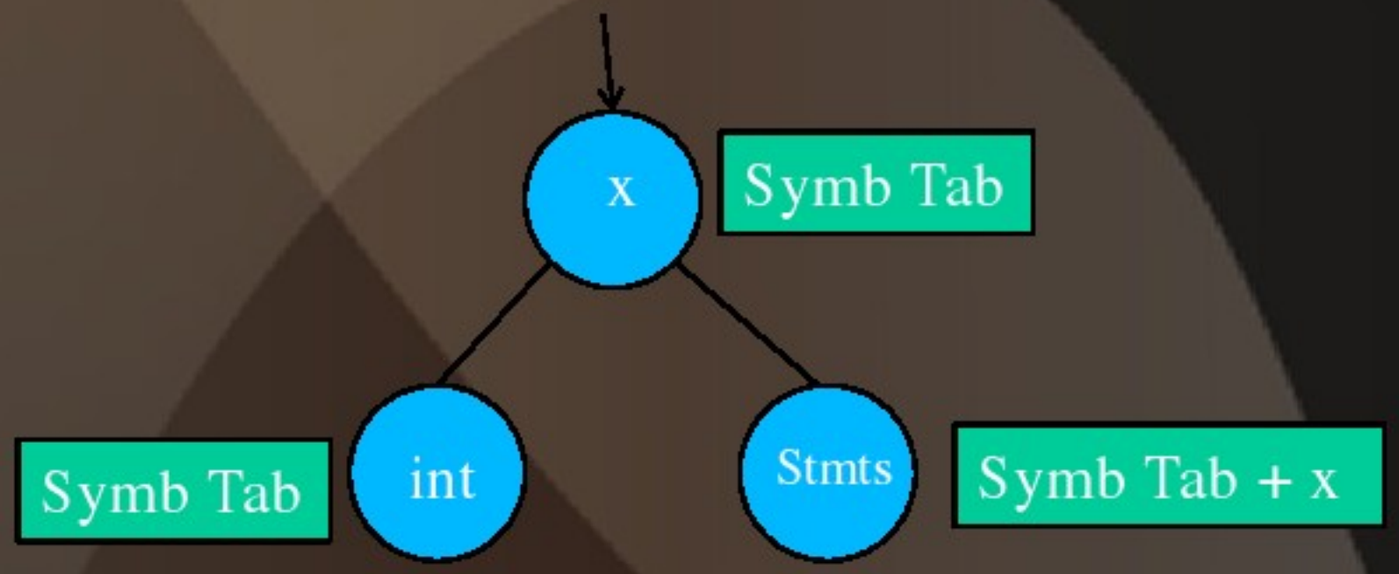
CSE346: Compilers, IIT Guwahati

Symbol Tables

- A generic algorithm used in much of semantic analysis can be expressed as a recursive descent of an AST
- When processing a particular node in the tree, do the following:
  - *Before: Process an AST node n*
  - *Recurse: Process the children of n*
  - *After: Finish processing the AST node n*
- When performing semantic analysis on a portion of the the AST, we need to know which identifiers are defined
  - *This scope resolution is done using symbol tables*

Symbol Tables

- Example: The scope of "Expression" bindings is one subtree of the AST:
- We have a code block: { int x; …stmts…; }



- **x** is defined in subtree **stmts**

Symbol Tables

- { int x; …stmts…; }
- Idea:
  - Before processing *stmts,* add definition of *x* to current definitions, overriding any other definition of *x*
  - Recurse
  - After processing *stmts,* remove definition of *x* and restore old definition of *x*
- *A symbol table is a data structure that tracks the current bindings of identifiers*

Symbol Tables

- For a simple symbol table we can use a stack
- Operations
  - add_symbol(x)
    - push x and associated info, such as x's type, on the stack
  - find_symbol(x)
    - search stack, starting from top, for x. Return first x found or NULL if none found
  - remove_symbol()
    - pop the stack

Symbol Tables

- The simple symbol table works for cases where:
  - Symbols added one at a time
  - Declarations are perfectly nested
- Does not work for more complex cases:
  - Example: f (int x, int x) {…}
    - Functions introduce simultaneous definitions in the same scope
- Solution:
  - Instead of a single stack, build a stack of scopes

Symbol Tables

- Operations on the stack of scopes
  - enter_scope()
    - start a new nested scope
  - find_symbol(x)
    - finds current x (or null)
  - add_symbol(x)
    - add a symbol x to the table
  - check_scope(x)
    - true if x defined in current scope
  - exit_scope()
    - exit current scope

Symbol Tables

- What will be the configuration of the scope stack when inside function C().

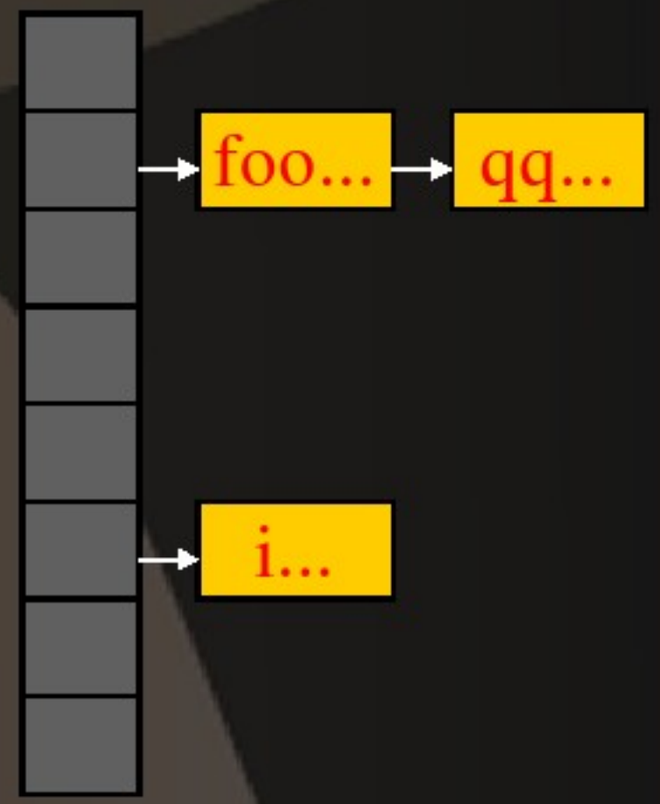
```
Class prog {
   int w;
   int x;
   void A() {
        int w;
        int y;
        int z;
        …
        void B() {int x; …}
        void C() {int x; int y; …}
    } // End of A
   Void D() {int z; …}
}
```

Organising the symbol table

- Linear List:
  - Simple approach, has no fixed size; but inefficient: a lookup may need to traverse the entire list: this takes $O(n)$.
- Binary tree:
  - An unbalanced tree would have similar behaviour as a linear list (this could arise if symbols are entered in sorted order).
  - A balanced would take $O(log_2 n)$ probes per lookup (worst-case). Techniques exist for dynamically rebalancing trees.
- Hash table:
  - Uses a hash function, h, to map names into integers; this is taken as a table index to store information. Potentially $O(1)$, but needs inexpensive function, with good mapping properties, and a policy to handle cases when several names map to the same single index.

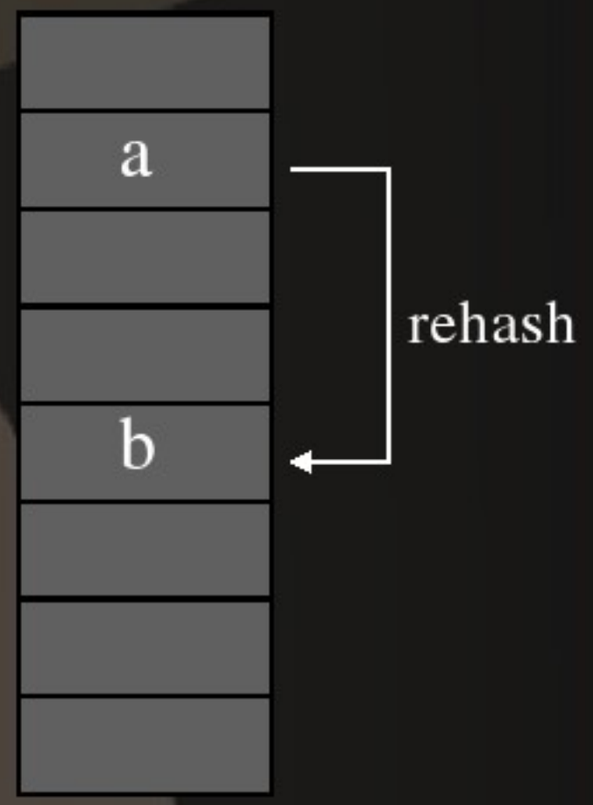
Bucket hashing (open hashing)

- A hash table consisting of a fixed array of *m* pointers to table entries.
- Table entries are organised as separate linked lists called buckets.
- Use the hash function to obtain an integer from *0* to *m-1*.
- As long as *h* distributes names fairly uniformly (and the number of names is within a small constant factor of the number of buckets), bucket hashing behaves reasonably well.



Linear Rehashing (open addressing)

- Use a single large table to hold records. When a collision is encountered, use a simple technique (i.e., add a constant) to compute subsequent indices into the table until an empty slot is found or the table is full. If the constant is relatively prime to the table size, this, eventually, will check every slot in the table.
- Disadvantages: too many collisions may degrade performance. Expanding the table may not be straightforward.



If $h(a)=h(b)$ rehash (say, add 3).

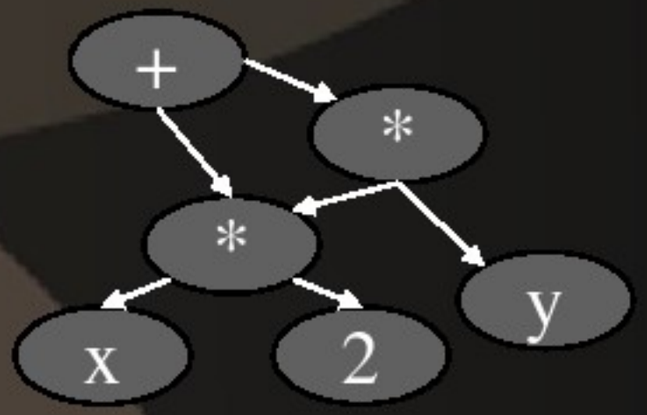
Symbol Tables

- Class names can be used before being defined
- We can't check class names
  - using a symbol table
  - or even in one pass
- Solution
  - Pass 1: Gather all class names
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes
  - Probably more than two

Directed Acyclic Graphs (DAGs)

A DAG is an AST with a unique node for each value

Example: The DAG for x*2+x*2*y



Powerful representation, encodes redundancy; but difficult to transform, not useful for showing control-flow.

Construction:

- Replace constructors used to build an AST with versions that remember each node constructed by using a table.
- Traverse the code in another representation.

Exercise: Construct the DAG for x=2*y+sin(2*x); z=x/2

Auxiliary Graph Representations

The following can be useful for analysis:

- **Control-Flow Graph** (CFG): models the way that the code transfers control between blocks in the procedure.
  - Node: a single basic block (a maximal straight line of code)
  - Edge: transfer of control between basic blocks.
  - (Captures loops, if statements, case, goto).

- **Data Dependence Graph**: encodes the flow of data.
  - Node: program statement
  - Edge: connects two nodes if one uses the result of the other
  - Useful in examining the legality of program transformations

- **Call Graph**: shows dependences between procedures.
  - Useful for inter-procedural analysis.

Control Flow Graphs

- **Control-Flow Graph** Example:

```
0: (A) t0 = read_num
1: (A) if t0 mod 2 == 0
2: (B)     print t0 + " is even."
3: (B)     goto 5
4: (C) print t0 + " is odd."
5: (D) end program
```

Find the basic blocks and draw the corresponding CFG

Examples

Draw the data dependence graph of:

```
1.  sum=0
2.  done=0
3.  while !done do
4.   read j
5.   if (j>0)
6.       sum=sum+j
7.       if (sum>100)
8.           done=1
9.   else
10.      sum=sum+1
11. endif
12.endif
13.endwhile
14.print sum
```

Examples

Draw the call graph of:

```
void a() { … b() … c() … f() … }
void b() { … d() … c() … }
void c() { … e() … }
void d() { … }
void e() { … b() … }
void f() { … d() …}
```