Semantic Analysis

- Type checking is an important semantic analysis technique
- What other semantic analyses might we want to do?
- Type checking (and code generation as we'll see later on) depends on an environment \( E \). How do we implement \( E \)?

Symbol tables

A symbol table associates values or attributes (e.g., types and values) with names.

What should be in a symbol table?
- variable names
- procedure and function names
- literal constants and strings
- source text labels (targets of GOTOS)

What information might compiler need?
- textual name
- data type
- dimension information (for arrays)
- lexical level of declaration
- storage class (base address)
- offset in storage
- if record, field offsets and lengths
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions/procedures

Implementation techniques

Unordered list
- use an array or linked list
- \( O(n) \) time search (\( n \) is number of entries)
- \( O(1) \) time insertion
- simple & compact, but too slow in practice

Binary search tree
- \( O(\log n) \) time expected search, \( O(n) \) worst case
- \( O(\log n) \) time expected insertion
- less simple approach

Hash table
- \( O(1) \) time expected for search
- \( O(1) \) time expected for insertion
- worst case is very unlikely
- subtle design issues, more programmer effort
- this approach is taken in most compilers
Binary search tree

Complexity
- O(log n) time expected search, O(n) worst case
- O(log n) time expected insertion
- O(n) space

Balanced Trees
- With a balanced tree, we get the expected times.
- On an arbitrary input, the tree is not necessarily balanced. What if the variables are alphabetized?
- Use an approximate balancing algorithm
  - If the height of sibling trees will vary by more than 1 after insertion, balance first by moving the deeper subtree to the shallower sibling.
  - Affects insertion performance only slightly
  - Complicates implementation
  - Space overhead is directly proportional to the number of items in the table.

Hash table

Key issues
- hashing function \( h(s) \rightarrow index \)
- table size \( m \) should be prime \( \) (seems to work best)
- how to pick new index?

Hash Functions: pick \( h(s) \) such that
- \( h(s) \) depends only on \( s \)
- \( h(s) \) is cheap to compute
- \( h(s) \) is uniform – each output is equally likely
- \( h(s) \) is randomizing – similar names do not map to similar values.
  Names may be user defined or compiler generated.

Sample hash functions \( (s = c_1c_2 \ldots c_k) \):
- \( (c_1 + c_2 + \ldots + c_k) \mod m \)
- \( (c_1 \times c_2 \times \ldots \times c_k) \mod m \)
- So that all bits of input affect output, avoid modulus by powers of 2.

Hash table

Complexity
- O(1) time expected for search
- O(1) time expected for insertion
- O(m) space (open addressing), for table size \( m \)
- O(n + m) space (bucket hashing or open hashing), where \( m \) is the number of slots and \( n \) is the number of hash table entries

open addressing – all elements are stored in the hash table itself.

Lookup and Insertion for open addressing:
1. Hash into an \( index \)
2. If Table[\( index \)] is empty
   - (a) \( lookup \) fails
   - (b) \( insertion \) adds at index
3. If Table[\( index \)] is full
   - (a) match implies \( lookup \) succeeds
   - (b) no match or \( insertion \) implies
     - pick new index and goto step 2 (full table?)

Collision resolution

A collision occurs when \( h(s_1) = h(s_2) \), but \( s_1 \neq s_2 \).

Collisions are inevitable unless the input is known in advance, in which case a perfect hash function can be found.

Linear Resolution (a.k.a. linear probing)
- If \( h(s) = k \) is full, try \( (k + 1) \mod m \). If it is full, try \( (k + 2) \mod m \), and so on.
- simple
- tends to build long chains (a.k.a. primary clustering)
Collision resolution (cont.)

Add-the-hash rehash
- If \( h(s) = k \) is full, try \( (2 \times h(s)) \mod m \). If it is full, try \( (3 \times h(s)) \mod m \) and so on.
- \( m \) must be prime,
- reduces primary clustering and secondary clustering (distributed chains of items that have the same hash code)

Quadratic rehash
- \( h(s), (h(s) + 1) \mod m, (h(s) + 2^2) \mod m, (h(s) + 3^2) \mod m, \ldots \)

Secondary hash functions
- \( h(s), (h(s) + h'(s)) \mod m, (h(s) + 2 \times h'(s)) \mod m, \ldots \)

Two-level hash table

Example \( \ldots \text{foo} \ldots \text{bar} \ldots \text{x} \ldots \)

Bucket hashing

Another approach to hash conflict resolution
- combines sparse index and lists of items
- hash item into one of the buckets
- walk list of items in bucket
- if item not found, add item to bucket

Bucket hashing complexity
- \( n \) number of names (table entries), \( m \) number of buckets
- \( \text{Avg search time} = 1/2 \times n/m \)
- \( \text{Avg insertion time} = n/m \) (insert if not yet there)

For \( n \leq km \) with fixed \( k \), time is \( O(k) = O(1) \).
- For large \( n \), a better data structure can improve performance. However, this is rarely necessary.
- \( O(m + n) \) space, but overhead of \( m \) is pretty small
- supports deletion
- not difficult to program

How hard is it to delete an entry in open addressing?

Bucket hashing

Can reorganize items in buckets

Scheme 1
*On each lookup, move item to front of bucket list*
- capitalize on locality, if possible
- reduce average case search

Scheme 2
*On each lookup, move item up by one position*
- capitalize on locality, if possible
- limit impact of a single lookup
- reduce average case search

Rivest and Tarjan showed that Scheme 2 does as well as any reorganizing scheme.
Block structure symbol tables

Nested lexical scoping - relative to the current component such as a procedure, module, statement, . . .

• current scope - the innermost scope for the current component
• open scope - all scopes surrounding the current scope
• closed scope - all other scopes

Which variables are visible?

• Only variables declared in the open scopes are visible
• Definitions of the same name in an inner scope take precedence over any outer scope
• New declarations are only made to the current scope
  ⇒ names in closed scopes are inaccessible

Nested scopes

What information is needed?

when we ask about a name, we want the most recent declaration
the declaration may be from the current procedure or some nested procedure

What operations do we need?

• insert(name, p) — create record for name at level p
• lookup(name) — returns pointer or index
• delete(p) — deletes all names declared at level p

May need to preserve list of locals for the debugger

Nested lexical scope example

Example:

Procedure A
H, I, J: integer
begin
Procedure B
X, Y: real
begin
  . . .
end
Procedure C
H, L, M: character
begin
  . . .
end
Visible:
Inaccessible:

Nested scopes

Table per scope

Use one symbol table per scope.
Chain together in list based on level of nesting.
May use stack to store tables.

• insert(name, p) adds to the level p table
  It may need to create the level p table and add it to the chain
• lookup(name) walks chain of tables, looking in each for name. Starts at deepest nesting and works outwards. Returns first occurrence of name
• delete(p) throws away table for level p
  It must be the top table on chain
Nested scopes

Global table

Represent all symbols in one table.
Add nesting level to all items.

Approach: build on bucket hashing

- \texttt{insert(name,p)} adds \texttt{(name,p)} to the front of the bucket list. Chain together records declared at level \texttt{p}
- \texttt{lookup(name)} naturally finds lexically closest definition, since first occurrence of name is from the deepest scope
- \texttt{delete(p)} walks the level \texttt{p} chain
  It removes each level \texttt{p} item and fixes up the pointers

\textit{Chain reorganization is more complex, but doable}

String storage

Storing identifier strings in symbol table entries requires either
- variable sized entries, or
- hard small limit on size, or
- much wasted space.

\implies \text{Store character strings in a string space and refer to them with simple fixed size descriptors, or use dynamic memory allocation.}

Next lecture

Runtime Environments

Please read ASU Chapter 7.1 – 7.5