Symbol Tables

For compile-time efficiency, compilers often use a symbol table:

- associates lexical *names* (symbols) with their *attributes*

What items should be entered?

- variable names
- defined constants
- procedure and function names
- literal constants and strings
- source text labels
- compiler-generated temporaries

Separate table for structure layouts (types - field offsets and lengths)

Symbol Table Information

What kind of information might the compiler need?

- textual name
- data type
- dimension information (for aggregates)
- declaring procedure
- lexical level of declaration
- storage class (base address)
- offset in storage
- if record, pointer to structure table
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions

Symbol table organization

How should the table be organized?

Linear List

- $O(n)$ probes per lookup
- easy to expand — no fixed size
- one allocation per insertion

Ordered Linear List

- $O(log_2 n)$ probes per lookup using binary search
- insertion is expensive (to reorganize list)

Binary Tree

- $O(n)$ probes per lookup — unbalanced
- $O(log_2 n)$ probes per lookup — balanced
- easy to expand — no fixed size
- one allocation per insertion

Hash Table

- $O(1)$ probes per lookup — on average
- expansion costs vary with specific scheme

Hash Tables

What about the hash function?

Properties:

- $h(c_1 c_2 ... c_k)$ depends solely on $c_1 c_2 ... c_k$
- h computed quickly
- *uniform* — equal probability of all hash values
- *randomizing* — similar symbols have dissimilar hash values

Examples: for table size m, $h(c_1 c_2 ... c_k) =$

1. $(c_1 \times c_k)$ mod m
2. $(\sum_{i=1}^{k} c_i)$ mod m
3. $(\prod_{i=1}^{k} c_i)$ mod m
4. h_k where $h_0 = 0$ and $h_i = \alpha h_{i-1} + c_i, 1 \leq i \leq k, \alpha$ prime

Hash Tables: Resolving Collisions

Linear resolution

- try $(h(c_1 c_2 ... c_k) + i)$ mod $m, i = 1, 2, 3, ...$
- problem: long chains as table fills

Add-the-hash rehash

- try $i \times h(c_1 c_2 ... c_k)$ mod $m, i = 2, 3, ...$
- prevents long chains, but m must be prime to eventually cover all hash values

Quadratic rehash

- try $(h(c_1 c_2 ... c_k) + i^2)$ mod $m, i = 1, 2, 3, ...$

Chaining (bucket hash table)

- minimizes table space overhead
- graceful performance degradation as table fills

Bucket Hash Table

Combine sparse index and a linear list

Lookup and Insertion

1. hash into one of m buckets
2. walk the bucket's list checking for item
3. if not found, add to front of list

Average case complexity – n elements, m buckets

- *lookup* — walk half the list $O(1 + \frac{n}{2m})$
- *insertion* — walk the entire list $O(2 + \frac{n}{m})$

Can we improve on the linear search?

Bucket Hash Table

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

Linear Rehash Table

Use simple linear table and rehash on collision

Lookup and Insertion

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?)

Key issues

- Step 3b — simply add k to index
- table size should be prime (at least odd)
- k and table size should be relatively prime

Linear Rehash Table

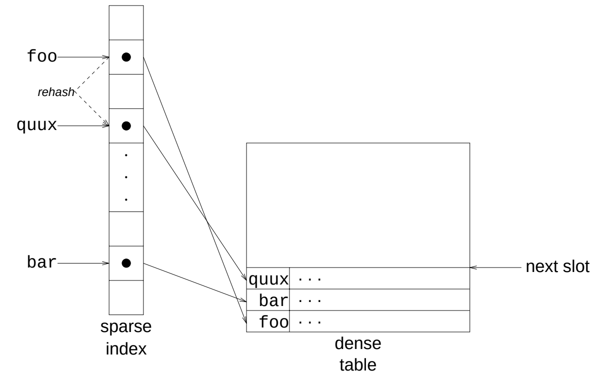
Scheme 1: Simple Table

- use a simple, sparse table
- moderately large data structure
- fixed size table
- reallocation is terrible

Scheme 2: Complex Table

- use a sparse map
- use a dense table
- table growth is easy
- map growth and rehash is simple
- file I/O simplified

Example



Nested Scopes: Block-Structured Symbol Tables

What information is needed?

- when we ask about a name, we want the *most recent* declaration
- the declaration may be from the current scope or some enclosing scope
- innermost scope overrides declarations from outer scopes

Key point: new declarations (usually) occur only in current scope

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 Scopes

Idea 1: Chain together procedure local hash 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
  Returns first occurence of $name$
- $delete(p)$ throws away table for level p
  It must be the top table on chain

Nested Scopes

Idea 2: Build on a bucket hash organization

- $insert(name, p)$ adds $(name, p)$ to the front of the bucket list
  Chain together records declared at level p

- $lookup(name)$ naturally finds lexically closest definition

- $delete(p)$ walks the level p chain
  It removes each level p item and fixes up the pointers

Chain reorganization is more complex, but doable

Nested Scopes

Idea 3: Build on a linear rehash scheme

- $insert(name, p)$ hashes by $name$.
  1. If $name$ isn't found, add it.
  2. If $name$ is there with wrong level,
     (a) create hidden name record
     (b) hang it off table slot
     (c) supersede information in active slot
  3. Add $name$ to level p chain

- $lookup(name)$ works without change

- $delete(p)$ walks the level p chain for each $name$ on the chain
  1. update the active record from front of chain
  2. deletes the first hidden name record from chain

Nested Scopes: Complications

Fields and records — either give each record type its own symbol table *or* assign record numbers to qualify field names in symbol table

with R **do** `<stmt>`:
- all IDs in `<stmt>` are treated first as R.id
- separate record tables — chain R's scope ahead of outer scopes
- record numbers — either open new scope, copy entries with R's record number *or* chain record numbers: search using these first

Implicit declarations:
- labels — declare and define name
- Ada/Modula-3/Tiger `FOR` loop: loop index has type of range specifier

Overloading:
- link alternatives (check no clashes), choose based on context

Forward references:
- bind symbol only after all possible definitions $\Rightarrow$ multiple passes

Other complications:
- packages, modules, interfaces — `IMPORT`, `EXPORT`

Attribute Information

Attributes are internal representation of declarations

Symbol table associates names with attributes

Names may have different attributes depending on their meaning:

- variables: type, procedure level, frame offset
- types: type descriptor, data size/alignment
- constants: type, value
- procedures: formals (names/types), result type, block information (local decls.), frame size

Type Expressions

Type expressions are a textual representation for types:

1. basic types: $boolean$, $char$, $integer$, $real$, etc.

2. type names

3. constructed types (constructors applied to type expressions):

   (a) arrays: $array(I, T)$ denotes array of elements of type T, index type I
   e.g., $array(1..10, integer)$

   (b) products: $T_1 \times T_2$ denotes the Cartesian product of type expressions T_1 and T_2

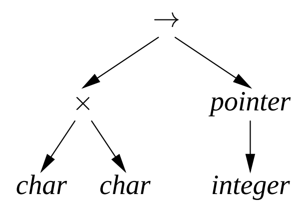
   (c) records: fields have names
   e.g., $record((\mathtt{a} \times integer), (\mathtt{b} \times real))$

   (d) pointers: $pointer(T)$ denotes the type "pointer to an object of type T"

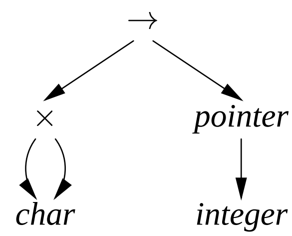
   (e) functions: $D \rightarrow R$ denotes type of a function mapping domain type D to range type R
   e.g., $integer \times integer \rightarrow integer$

Type Descriptors

Type descriptors are compile-time structures representing type expressions
e.g., $char \times char \rightarrow pointer(integer)$



or



Type Compatibility

Type checking needs to determine type equivalence

Two approaches:

Name equivalence: each type name is a distinct type

Structural equivalence: two types are equivalent iff. they have the same structure (after substituting type expressions for type names)

- $s \equiv t$ iff. s and t are the same basic types
- $array(s_1, s_2) \equiv array(t_1, t_2)$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $s_1 \times s_2 \equiv t_1 \times t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $pointer(s) \equiv pointer(t)$ iff. $s \equiv t$
- $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$

Type Compatibility: Example

Consider:

```
type link  = ↑cell;
var  next  : link;
     last  : link;
     p     : ↑cell;
     q, r  : ↑cell;
```

Under name equivalence:

- `next` and `last` have the same type
- `p`, `q` and `r` have the same type
- `p` and `next` have different type

Under structural equivalence all variables have the same type

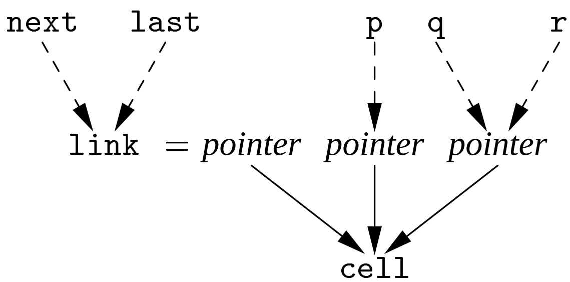
Ada/Pascal/Modula-2/Tiger are somewhat confusing: they treat distinct type definitions as distinct types, so:

`p` has different type from `q` and `r`

Type Compatibility: Pascal-Style Name Equivalence

Build compile-time structure called a *type graph*:

- each constructor or basic type creates a node
- each name creates a leaf (associated with the type's descriptor)



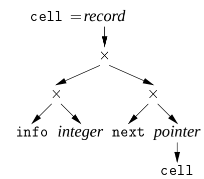
Type expressions are equivalent if they are represented by the same node in the graph

Type Compatibility: Recursive Types

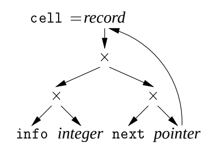
Consider:

```
type  link  =  ↑cell;
      cell  =  record;
               info  :  integer;
               next  :  link;
               end;
```

We may want to eliminate the names from the type graph. Eliminating name link from type graph for record:



Allowing cycles in the type graph eliminates cell:



Overloading

Most languages have type overloading:

- If nothing else, for integers/floats
- Equality/assignment overloaded for almost anything
- In languages with dynamic types (Lisp, Smalltalk), decision on what to do depends on type check at run-time
- Very inefficient for integers/floats
- Can be resolved at compile-time by *type inference*
- Type inference is usually done bottom-up
  - Say we have f can be either $int \rightarrow int$ or $float \rightarrow float$
  - Then $f(42)$ has only one valid typing: int

Polymorphic Functions

Polymorphism = many shapes

- Ad-hoc polymorphism: on a case-by-case basis; overloading
- Parametric polymorphism: can take a type as an argument
  - Templates
  - "True" parametric polymorphism:
    - ∗ function length(L) = if null(L) then 0 else 1+length tail(L)
    - ∗ length: $List(\alpha) \rightarrow int$
    - ∗ function first(L) = head(L)
    - ∗ first: $List(\alpha) \rightarrow \alpha$
    - ∗ function reverse(l) = ...
    - ∗ reverse: $List(\alpha) \rightarrow List(\alpha)$
  - Often combined with type inference