Semantic Analysis

- Recall : The compilation process must preserve the semantics of the original program.

- Semantics of a program can be context sensitive.

Copyright ©2000 by Antony L. Hosking. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@cs.purdue.edu.*

Context Sensitive Analysis

- Is v1 scalar, an array, or a function?
- Is v1 declared/defined before it is used?
- Are any names declared but not used?
- * Which declaration of v1 is used ?
- * Is an expression type-consistent ?
- Does the dimension of a reference match the declaration?
- Where v1 can be stored? (heap, stack)
- Does reference the result of a malloc()?
- Is an array reference in bounds?
- * Can a variable be replaced with a constant?
- * Does function produce a constant value?

These cannot be answered with a context-free grammar

Context Sensitve Analysis

Why is context sensitve analysis hard?

- Answers depend on values, not syntax.
- Questions deal with non-local data.
- Answers might impact non-local questions.
- Might requires some computation.

Symbol tables

For *compile-time* efficiency, compilers 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 (*we'll get there*)

Separate table for structure layouts (types)

(*field offsets and lengths*)

A symbol table is a compile-time structure

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

Nested scopes: block-structured symbol tables

What information is needed?

- when asking about a name, want *most recent* declaration
- declaration may be from current scope or outer scope
- innermost scope overrides outer scope declarations

Key point: new declarations occur only in current scope

What operations do we need?

- `void put (Symbol key, Object value)`
  bind key to value
- `Object get(Symbol key)`
  return value bound to key
- `void beginScope()`
  remember current state of table
- `void endScope()`
  close current scope and restore table to state at most recent open beginScope

May need to preserve list of locals for the debugger

Nested scopes: complications

Fields and records:

give each record type its own symbol table
or assign record numbers to qualify field names in 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:
  open new scope, copy entries with R's record number
  or chain record numbers: search using these first

Nested scopes: complications (cont.)

Implicit declarations:

- labels:
  declare and define name (in Pascal accessible only within enclosing scope)
- 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

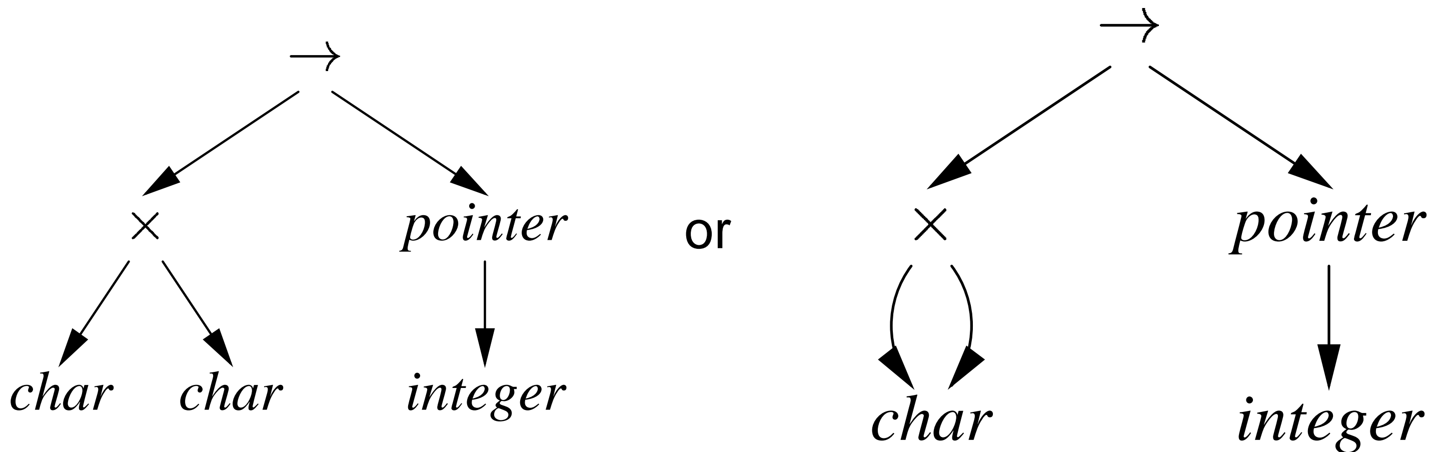
Type expressions are a textual representation for types:

1. basic types: $boolean$, $char$, $integer$, $real$, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   (a) $array(I,T)$ denotes an array of T indexed over I
       e.g., $array(1 \dots 10, integer)$
   (b) products: $T_1 \times T_2$ denotes 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 the 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., $char \times char \rightarrow pointer(integer)$



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

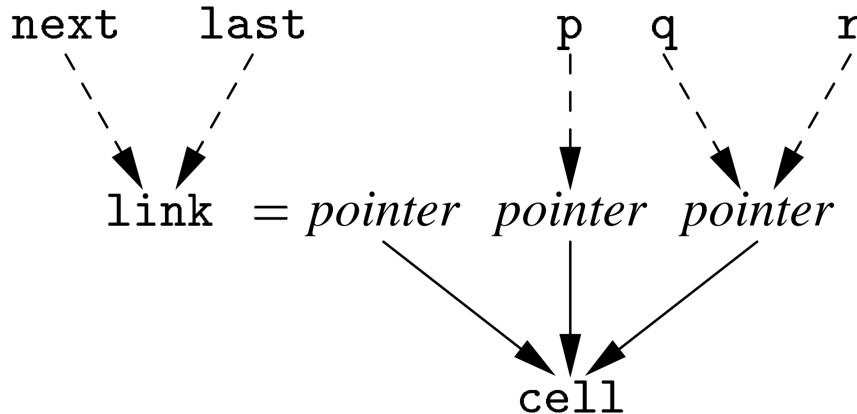
Under structural equivalence all variables have the same type

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 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

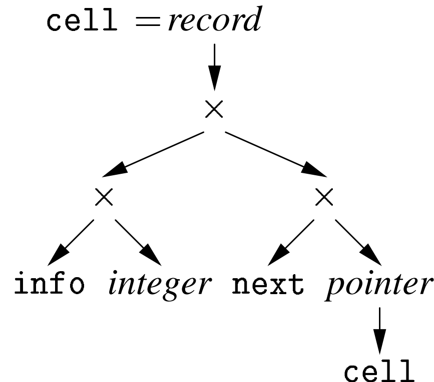
Type compatibility: recursive types

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:



Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`: