Semantic Analysis

The compilation process is driven by the syntactic structure of the program as discovered by the parser

Semantic routines:

- interpret meaning of the program based on its syntactic structure
- two purposes:
  - finish analysis by deriving context-sensitive information
  - begin synthesis by generating the IR or target code
- associated with individual productions of a context free grammar or subtrees of a syntax tree

Context-sensitive analysis

What context-sensitive questions might the compiler ask?

1. Is `x` scalar, an array, or a function?
2. Is `x` declared before it is used?
3. Are any names declared but not used?
4. Which declaration of `x` does this reference?
5. Is an expression *type-consistent*?
6. Does the dimension of a reference match the declaration?
7. Where can `x` be stored? (heap, stack, ...)
8. Does `*p` reference the result of a malloc()?
9. Is `x` defined before it is used?
10. Is an array reference *in bounds*?
11. Does function `foo` produce a constant value?
12. Can `p` be implemented as a *memo-function*?

These cannot be answered with a context-free grammar

Context-sensitive analysis

Why is context-sensitive analysis hard?

- answers depend on values, not syntax
- questions and answers involve non-local information
- answers may involve computation

Several alternatives:

abstract syntax tree (*attribute grammars*)	specify non-local computations automatic evaluators
symbol tables	central store for facts express checking code
language design	simplify language avoid problems

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

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 (usually) 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

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

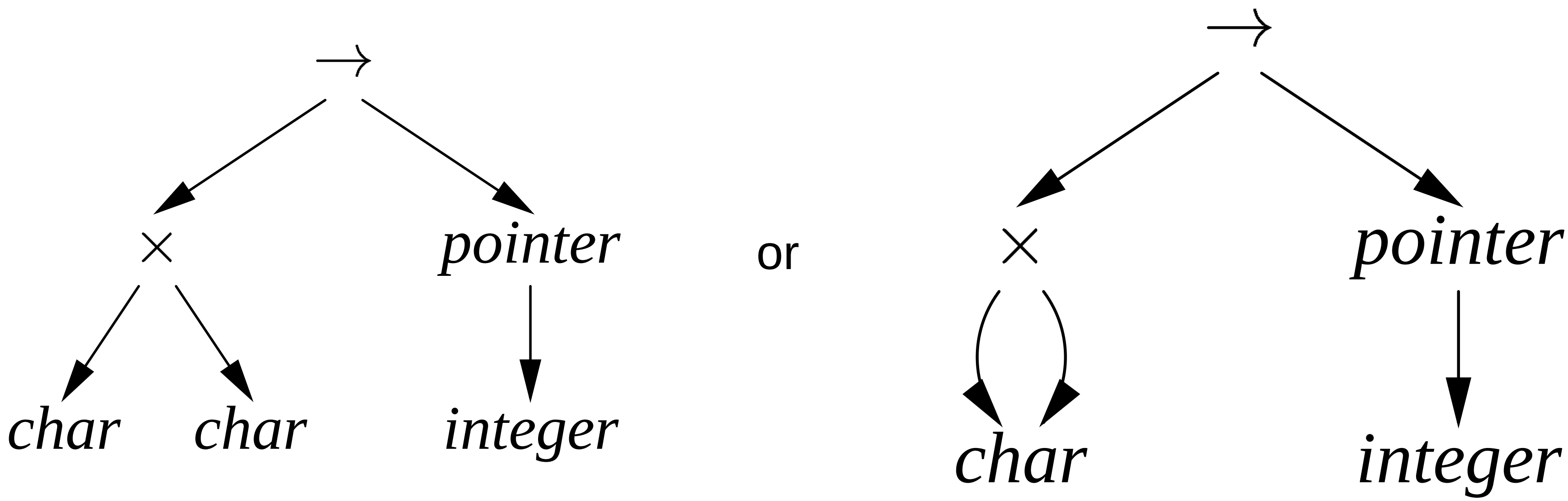
Type expressions are a textual representation for types:

1. basic types: $boolean$, $char$, $integer$, $real$, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   (a) $array(I, T)$ denotes array of elements type T, index type I
   e.g., $array(1 \ldots 10, integer)$
   (b) $T_1 \times T_2$ 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) $pointer(T)$ denotes the type “pointer to object of type T”
   (e) $D \rightarrow R$ denotes type of function mapping domain D to range 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. $s \equiv t$
- $s_1 \to s_2 \equiv t_1 \to t_2$ 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

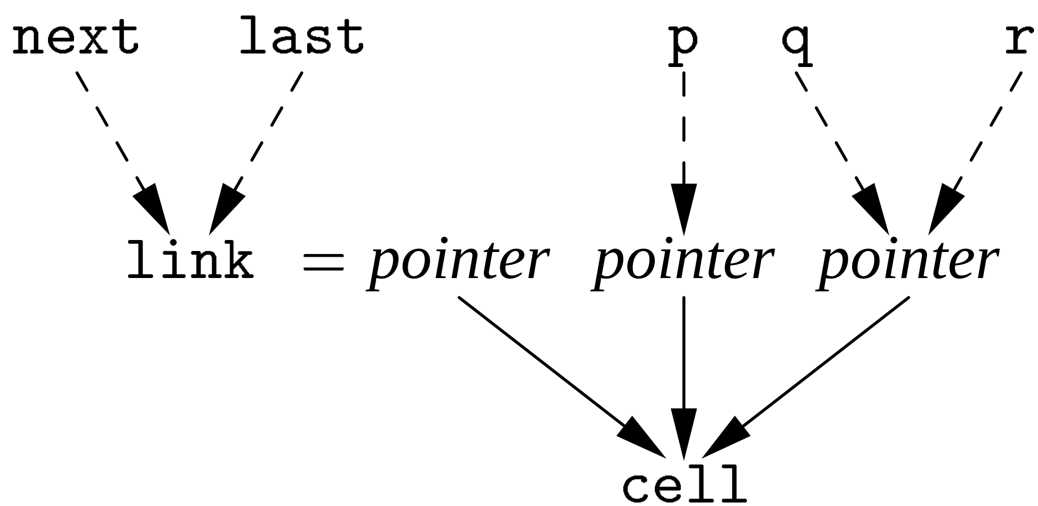
Ada/Pascal/Modula-2/Tiger are somewhat confusing: they treat distinct type definitions as distinct types, so

`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

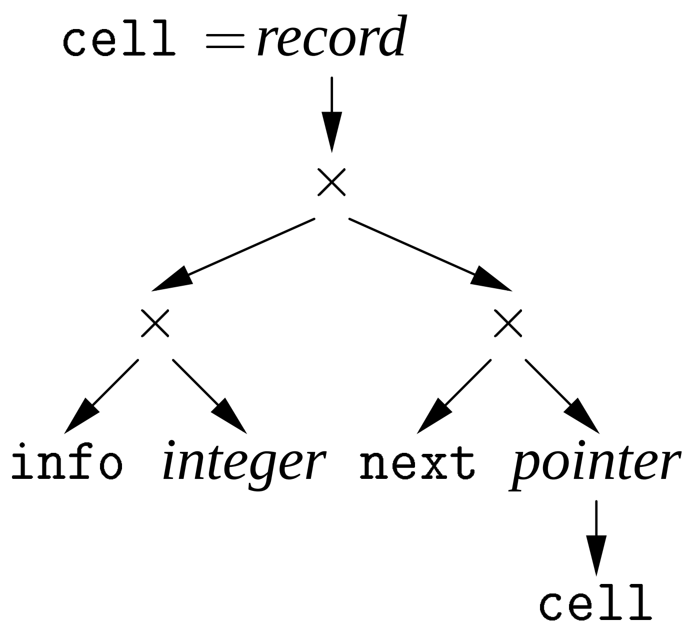
Type compatibility: recursive types

Consider:

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

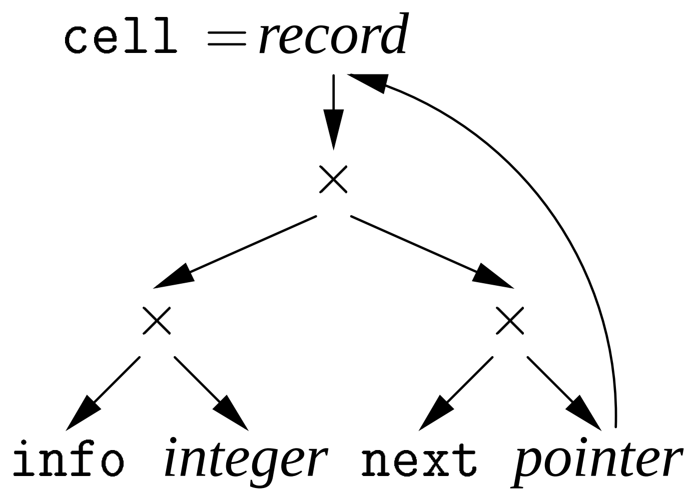
We may want to eliminate the names from the type graph

Eliminating name `link` from type graph for record:



Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`:



Java inheritance: field overloading

- Fields declared in a subclass can *overload* fields declared in superclasses
- Overloading is same name used in different contexts to refer to different things, such as *different* fields
- Consider:

```
class A { int j; }
class B extends A { int j; }

A a = new A();// let's call this object X
              // X has one field, named j, declared in A
a.j = 1;      // assigns 1 to the field j of X declared in A
a = new B();  // let's call this object Y
              // Y has two fields, both named j,
              // one declared in A, the other in B
a.j = 2;      // assigns 2 to the field j of Y declared in A
B b = a;
b.j = 3;      // assigns 3 to the field j of Y declared in B
```

Java inheritance: method overriding

- Methods declared in subclasses can *override* methods declared in superclasses
- Overriding is same name used to name a different thing, regardless of context, such as methods in subclasses with the same name
- Consider:

```
class A { int j; void set_j(int i) { this.j = i; }
class B extends A { int j; void set_j(int i) { this.j = i; }

A a = new A();// let's call this object X
a.set_j(1);   // assigns 1 to the field j of X declared in A
              // i.e., invokes A set_j method
a = new B();  // let's call this object Y
a.set_j(2);   // assigns 2 to the field j of Y declared in B
              // i.e., invokes B set_j method
B b = a;
b.set_j(3);   // assigns 3 to the field j of Y declared in B
              // i.e. invokes B set_j method
```

Java method overloading

- Java also supports method overloading, which has nothing to do with inheritance
- Consider:

```
class A {
  int j;
  boolean b;
  void set(int i) { this.j = i; }
  void set(boolean b) { this.j = b; }
}
```

- Don't confuse method overloading with method overriding