CS3300 - Compiler Design

Intro to Semantic Analysis

V. Krishna Nandivada

IIT Madras

Acknowledgement

Semantic Processing

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 (e.g. type checking)
  - begin synthesis by generating the IR or target code
- associated with individual productions of a context free grammar or subtrees of a syntax tree

Alternatives for semantic processing

- one-pass analysis and synthesis
- one-pass compiler plus peephole
- one-pass analysis & IR synthesis + code generation pass
- multipass analysis (e.g. gcc)
- multipass synthesis (e.g. gcc)
- language-independent and retargetable (e.g. gcc) compilers

Our focus in the assignments: One-pass analysis & IR synthesis + multipass analysis + multipass synthesis.

Goal - Type checking (MiniJava)

- We need generate type information.
  - For fields, variables, expressions, functions.
- Need to enforce types:
  - Assignments, function calls, expressions.
- We need to remember the type information and recall them as/where required – symbol table.

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)

A symbol table is a compile-time structure
Separate table for structure layouts (types) (includes field offsets and lengths)
May need to preserve list of locals for the debugger

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
- . . .

Storage classes of variables

During code generation, each variable is assigned an address (addressing method), approrpriate to its storage class.

- A local variable is not assigned a fixed machine address (or relative to the base of a module) – rather a stack location that is accessed by an offest from a register whose value does not point to the same location, each time the procedure is invoked. Why is it interesting?
- Four major storage classes: global, stack, stack static, registers

Symbol table organization

How should the table be organized?

- Linear List
  - $\mathbf{O}(n)$ probes per lookup
  - easy to expand — no fixed size
  - one allocation per insertion
- Ordered Linear List
  - $\mathbf{O}(\log_2 n)$ probes per lookup using binary search
  - insertion is expensive (to reorganize list)
- Binary Tree
  - $\mathbf{O}(n)$ probes per lookup — unbalanced
  - $\mathbf{O}(\log_2 n)$ probes per lookup — balanced
  - easy to expand — no fixed size
  - one allocation per insertion
- Hash Table
  - $\mathbf{O}(1)$ 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

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 ⇒ 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

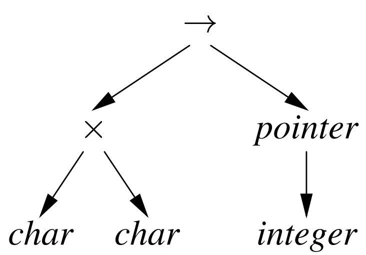
Type expressions

Type expressions are a textual representation for types:

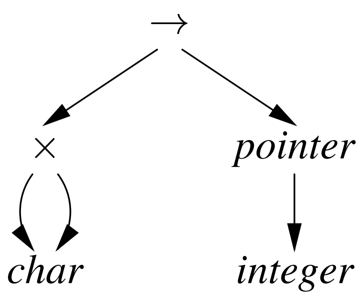
1. basic types: *boolean*, *char*, *integer*, *real*, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   1. $array(I,T)$ denotes an array of T indexed over I
      e.g., $array(1\dots10, integer)$
   2. products: $T_1 \times T_2$ denotes Cartesian product of type expressions T_1 and T_2
   3. records: fields have names
      e.g., $record((\mathtt{a} \times integer),(\mathtt{b} \times real))$
   4. pointers: $pointer(T)$ denotes the type "pointer to an object of type T"
   5. functions: $D \to R$ denotes the type of a function mapping domain type D to range type R
      e.g., $integer \times integer \to integer$

Type descriptors

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



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. $s \equiv t$
- $s_1 \to s_2 \equiv t_1 \to t_2$ 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

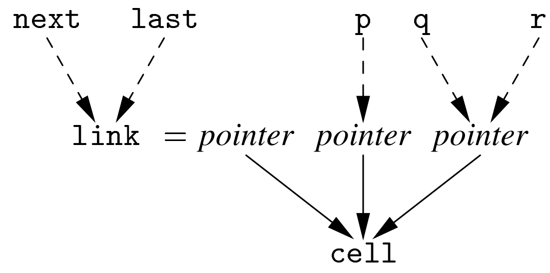
Under structural equivalence all variables have the same type

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 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

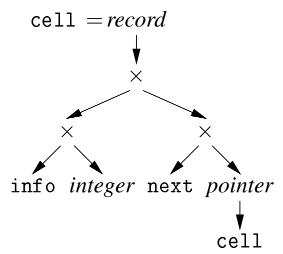
Type compatibility: recursive types

Consider:

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

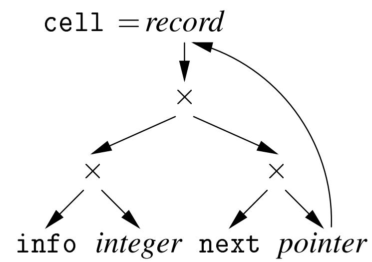
We may want to eliminate the names from the type graph

Eliminating name `link` from type graph for record:



Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`:



Think: If structural equivalence was to be used by Java, how to type check?

Enforcing type checks in MiniJava

Examples

- Assignment statements,
- If-expression,
- Arithmatic expression,
- Function call,
- Return statement,