Type Checking

Roadmap (Where are we?)

Last lecture

- Context-sensitive analysis
  - → Motivation
  - → Attribute grammars
  - → Ad hoc Syntax-directed translation

This lecture

- Type checking
  - → Type systems
  - → Using syntax directed translation
- Symbol tables
  - → Lexical scoping
  - → Implementation
    - Stack
    - Threaded stack

Types

Type: A set of values and meaningful operations on them

Types provide semantic "sanity checks" (consistency checks) and determine efficient implementations for data objects

Types help identify

- → errors, if an operator is applied to an incompatible operand
  - dereferencing of a non-pointer
  - adding a function to something
  - incorrect number of parameters to a procedure
  - …
- → which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)
- → identification of polymorphic functions

Type Systems

Type system: Each language construct (operator, expression, statement, …) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

Type expressions for

- → basic types
  - `integer, char, real, boolean, typeError`
- → constructed types // T is a type expression
  - `array(lb…ub, T)` // array of T
  - `pointer(T)` // pointer to T
  - T1 **x** T2 // tuple of T1, T2
  - T1 → T2 // function w/ arg T1 returning T2

Type Checker

- A type checker implements a type system. It computes or "constructs" type expressions for each language construct
- Static type checking
  - → Detects type errors at compile time
  - → No run time overhead
  - → Not always possible (e.g., A[i])
- Dynamic type checking
  - → Performed at run time
  - → More flexible, allows prototyping
  - → Run-time overhead to maintain & check tags

Types Inference Rules

- Specifies the type of an expression
- Example
  - → If operands of addition are of type integer, result is of type integer
  - → Result of unary & operator is pointer to type of operand
- Denotational semantics of type inference rule

$$\frac{E \quad e_1 : \text{integer} \quad E \quad e_2 : \text{integer}}{E \quad (e_1 + e_2) : \text{integer}}$$

where E is a type environment that maps constants and variables to their type expressions.

- Question
  - → How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

3.0 + 1 or 1 + 3.0

Type Equivalence

Structural -- type equivalence: type names are expanded
Name -- type equivalence: type names are not expanded

Example:

```
type A is  array(1..10) of integer;
type B is  array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence: (a, b, c, d, e)
name equivalence: (a), (b), (c, d, e)

Syntax Directed Translation Scheme (in CUP)

Revisit our type inference rule for "+".

```
exp  ::=  exp:e1 PLUS exp:e2
  {: if (e1 == sym.INT && e2 == sym.INT )
      RESULT = sym.INT ;
    else {
      RESULT = typeError;
      System.out.println("Error: illegal operand types");
    } :}
```

- The definition of type expression as Java types (static final int fields in class sym) should be done in mycc.cup.
- The assignment of type expression Java types to terminals and nonterminals of the grammar is done in mycc.cup.

Syntax Directed Translation Scheme (in Yacc)

Revisit our type inference rule for "+".

```
exp  : exp '+' exp  {  if ($1 == integer && $3 == integer)
                          $$ = integer;
                        else {
                          $$ = typeError;
                          printf("Error: illegal operand types\n");
                        } }
```

- The definition of type expression as C types (structs) should be done in attr.h . attr.c may contain helper functions.
- The assignment of type expression C types to terminals and nonterminals of the grammar is done in parse.y.

Type Checker Example

Grammar for source language:

P ::= D ; E
D ::= D ; E | id: T
T ::= char | integer | array [num] of T | $\uparrow$ T
E ::= literal | num | id | E mod E | E[E] | E $\uparrow$
T ::= T $\rightarrow$ T declaration
E ::= E (E) application

- Basic types *char, integer, typeError*
- assume all arrays start at 1, *e.g.,*
  array [256] of char
  results in the type expression $array(1\ldots256, char)$
- $\uparrow$ builds a pointer type, so $\uparrow$ integer
  results in the type expression $pointer(integer)$

Type Checker Example (cont.)

- Handling declarations

D ::= id: T		$\{\ addtype(\text{id}.entry,\ T.type)\ \}$
T ::= char		$\{\ T.type \leftarrow char\ \}$
T ::= integer		$\{\ T.type \leftarrow integer\ \}$
T ::= $\uparrow T_1$		$\{\ T.type \leftarrow pointer(T_1.type)\ \}$
T ::= array [num] of T_1		$\{\ T.type \leftarrow array(1\ldots\text{num}.val,\ T_1.type)\ \}$

Type Checker Example (cont.)

- Handling expressions

E ::= literal	$\{\ E.type \leftarrow char\ \}$
E ::= num	$\{\ E.type \leftarrow integer\ \}$
E ::= id	$\{\ E.type \leftarrow lookup(\text{id}.entry)\ \}$
E ::= E_1 mod E_2	$\{\ E.type \leftarrow$ if $E_1.type = integer$ and $E_2.type = integer$ then $integer$ else $typeError\ \}$
E ::= $E_1[E_2]$	$\{\ E.type \leftarrow$ if $E_2.type = integer$ and $E_1.type = array(s,t)$ then t else $typeError\ \}$
E ::= $E_1 \uparrow$	$\{\ E.type \leftarrow$ if $E_1.type = pointer$ then t else $typeError\ \}$

Type Checker Example (cont.)

- Handling statements

S ::= id ← E	{ $S.type \leftarrow$ if id.$type = E.type$ then $void$ else $typeError$ }
S ::= if E then S_1	{ $S.type \leftarrow$ if $E.type = boolean$ then S_1.type else $typeError$ }
S ::= while E do S_1	{ $S.type \leftarrow$ if $E.type = boolean$ then S_1.type else $typeError$ }
S ::= S_1 ; S_2	{ $S.type \leftarrow$ if $S_1.type = void$ then $void$ else $typeError$ }

Type Checker Example (cont.)

- Handling functions

T ::= $T_1 \rightarrow T_2$ { $T.type \leftarrow (T_1.type \rightarrow T_2.type)$ }

E ::= E_1 (E_2) { $E.type \leftarrow$ if $E_1.type = s \rightarrow t$ and $E_2.type = s$ then t else $typeError$ }

Symbol Tables

- Symbol table
  - → Compile-time structures for resolving references to names
    - Will look at run-time structures later
  - → Can also associate attributes with name
- Attributes possibly associated with name
  - → Type
  - → Declaring procedure
  - → Lexical level
  - → If array, number and size of dimensions
  - → If function, number and type of parameters

Lexically-scoped Symbol Tables § 5.7 in EaC

The problem
- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface
- insert(*name, level*) – creates record for *name* at *level*
- lookup(*name, level*) – returns pointer or index
- delete(*level*) – removes all names declared at *level*

Many implementation schemes have been proposed (see § B.4)
- We'll stay at the conceptual level
- Hash table implementation is tricky, detailed, & fun

Example

```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int x, y, z
            ....
        }
        procedure s {
            int x, a, v
            …
        }
        … r … s
    }
    … q …
}
```

```
B0: {
        int a, b, c
B1:     {
            int v, b, x, w
B2:             {
                int x, y, z
                ....
            }
B3:             {
                int x, a, v
                …
            }
            …
        }
        …
    }
```


Example

```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int x, y, z
            ....
        }
        procedure s {
            int x, a, v
            …
        }
        … r … s
    }
    … q …
}
```

c

b, w

x, a, v

B0: {
int a_0, b_1, c_2
B1: {
int v_3, b_4, x_5, w_6
B2: {
int x_7, y_8, z_9
....
}
B3: {
int x_{10}, a_{11}, v_{12}
…
}
…
}
…
}

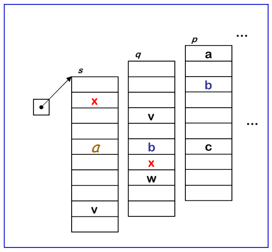
a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, no y or z

Picturing it as a series of Algol-like procedures

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



"Chain of tables" implementation

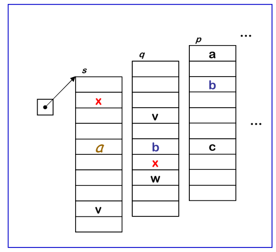
- *insert*() may need to create table
- it always inserts at current level
- *lookup*() walks chain of tables & returns first occurrence of name
- *delete*() throws away table for level *p*, if it is top table in the chain

Individual tables can be hash tables.

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



Remember

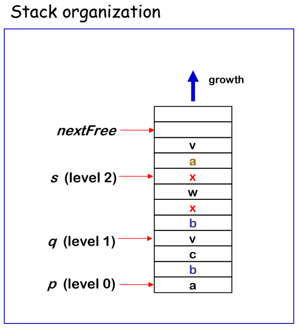
a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, no y or z

the names visible in s

If we add the subscripts, the relationship between the code and the table becomes clear

Implementing Lexically Scoped Symbol Tables

Stack organization



Implementation

- **insert () creates new level pointer if needed and inserts at nextFree**
- **lookup () searches linearly from nextFree-1 forward**
- **delete () sets nextFree to the equal the start location of the level deleted.**

Advantage

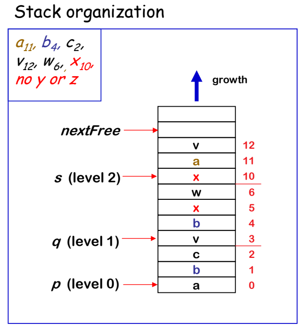
- **Uses much less space**

Disadvantage

- **Lookups can be expensive**

Implementing Lexically Scoped Symbol Tables

Stack organization



Implementation

- **insert () creates new level pointer if needed and inserts at nextFree**
- **lookup () searches linearly from nextFree-1 forward**
- **delete () sets nextFree to the equal the start location of the level deleted.**

Advantage

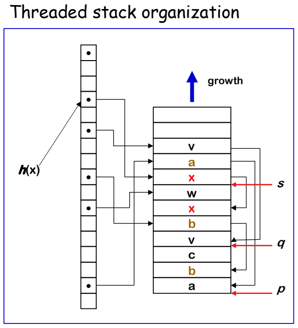
- **Uses much less space**

Disadvantage

- **Lookups can be expensive**

Implementing Lexically Scoped Symbol Tables

Threaded stack organization



Implementation

- **insert () puts new entry at the head of the list for the name**
- **lookup () goes direct to location**
- **delete () processes each element in level being deleted to remove from head of list**

Advantage

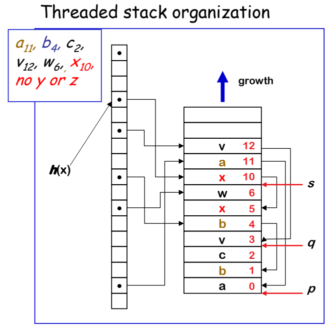
- **lookup is fast**

Disadvantage

- **delete takes time proportional to number of declared variables in level**

Implementing Lexically Scoped Symbol Tables

Threaded stack organization



Implementation

- **insert () puts new entry at the head of the list for the name**
- **lookup () goes direct to location**
- **delete () processes each element in level being deleted to remove from head of list**

Advantage

- **lookup is fast**

Disadvantage

- **delete takes time proportional to number of declared variables in level**