# Lecture 7: Type Systems and Symbol Tables

CS 540

George Mason University

# Static Analysis

Compilers examine code to find semantic problems.

- Easy: undeclared variables, tag matching
- Difficult: preventing execution errors

Essential Issues:

- Part I: Type checking
- Part II: Scope
- Part III: Symbol tables

# Part I: Type Checking

# Type Systems

- A **type** is a set of values and associated operations.
- A **type system** is a collection of rules for assigning type expressions to various parts of the program.
  - Impose constraints that help enforce correctness.
  - Provide a high-level interface for commonly used constructs (for example, arrays, records).
  - Make it possible to tailor computations to the type, increasing efficiency (for example, integer vs. real arithmetic).
  - Different languages have different type systems.

# Inference Rules - Typechecking

- Static (compile time) and Dynamic (runtime).
- One responsibility of a compiler is to see that all symbols are used correctly (i.e. consistently with the type system) to prevent execution errors.
- Strong typing – All expressions are guaranteed to be type consistent although the type itself is not always known (may require additional runtime checking).

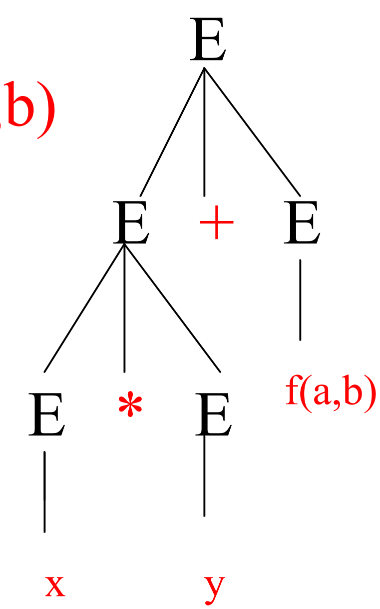
# What are Execution Errors?

- Trapped errors – errors that cause a computation to stop immediately
  - Division by 0
  - Accessing illegal address
- Untrapped errors – errors that can go unnoticed for a while and then cause arbitrary behavior
  - Improperly using legal address (moving past end of array)
  - Jumping to wrong address (jumping to data location)
- A program fragment is safe if it does not cause untrapped errors to occur.

# Typechecking

We need to be able to assign types to all expressions in a program and show that they are all being used correctly.

Input: x * y + f(a,b)




- Are x, y and f declared?
- Can x and y be multiplied together?
- What is the return type of function f?
- Does f have the right number and type of parameters?
- Can f's return type be added to something?

# Program Symbols

- User defines symbols with associated meanings. Must keep information around about these symbols:
  - Is the symbol declared?
  - Is the symbol visible at this point?
  - Is the symbol used correctly with respect to its declaration?

# Using Syntax Directed Translation to process symbols

While parsing input program, need to:

- Process declarations for given symbols
  - Scope – what are the visible symbols in the current scope?
  - Type – what is the declared type of the symbol?
- Lookup symbols used in program to find current binding
- Determine the type of the expressions in the program

# Syntax Directed Type Checking

Consider the following simple language

P → D S

D → id: T ; D | ε

T → integer | float | array [ num ] of T | ^T

S → S ; S | id := E

E → int_literal | float_literal | id | E + E | E [ E ] | E^

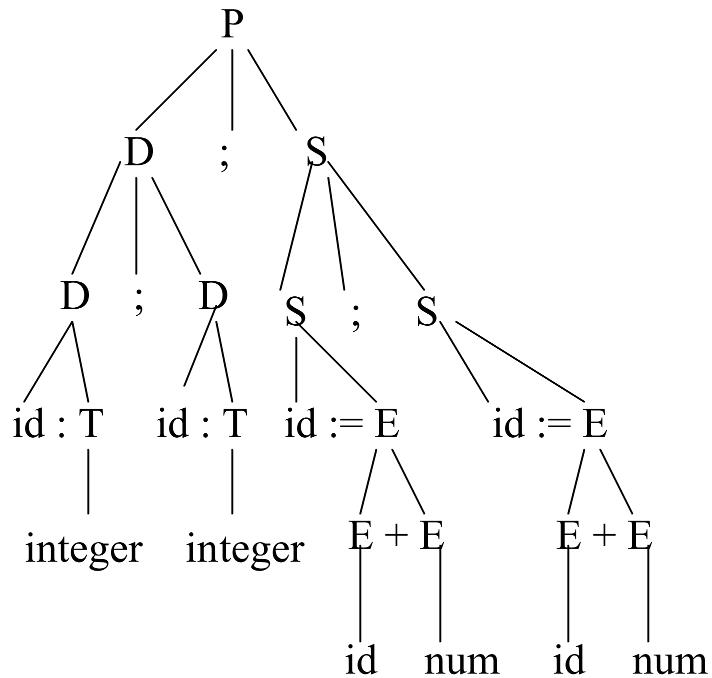
How can we typecheck strings in this language?

# Example of language:

i: integer; j: integer

i := i + 1;

j := i + 1



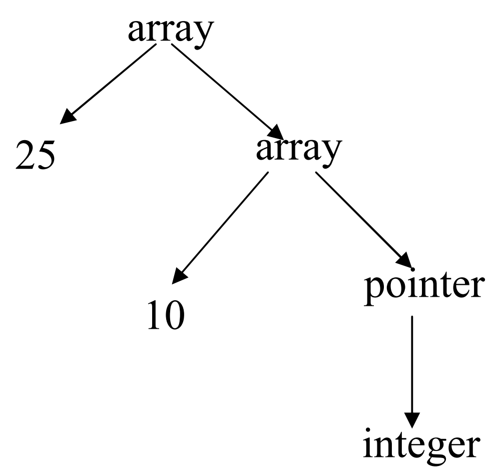
# Processing Declarations

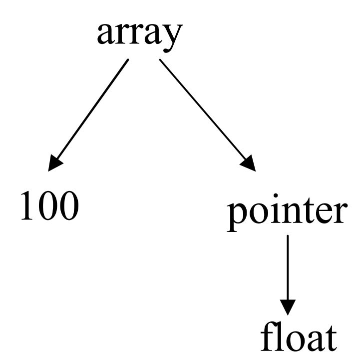
Put info into
the symbol table

| | |
|---|---|
| D → id : T ; D | {insert(id.name,T.type);} |
| D → ε | |
| T → integer | {T.type = integer;} |
| T → float | {T.type = float;} |
| T → array [ num ] of $T_1$ | {T.type = array($T_1$.type,num); } |
| T → ^$T_1$ | {T.type = pointer($T_1$.type);} |

Accumulate information about
the declared type

# Can use Trees (or DAGs)  to Represent Types




array[25]  of array[10] of  ^(integer)

array[100] of ^(float)

Build data structures while we parse

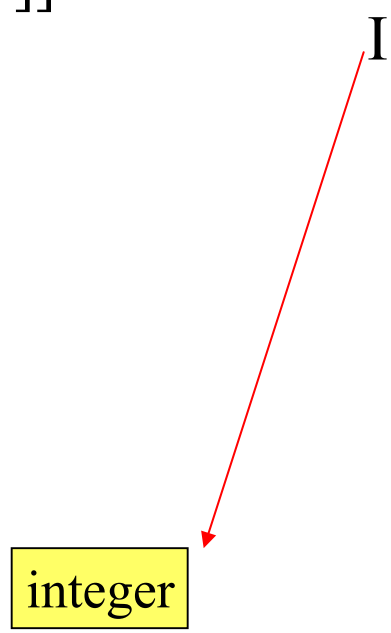
# Example

I: integer;

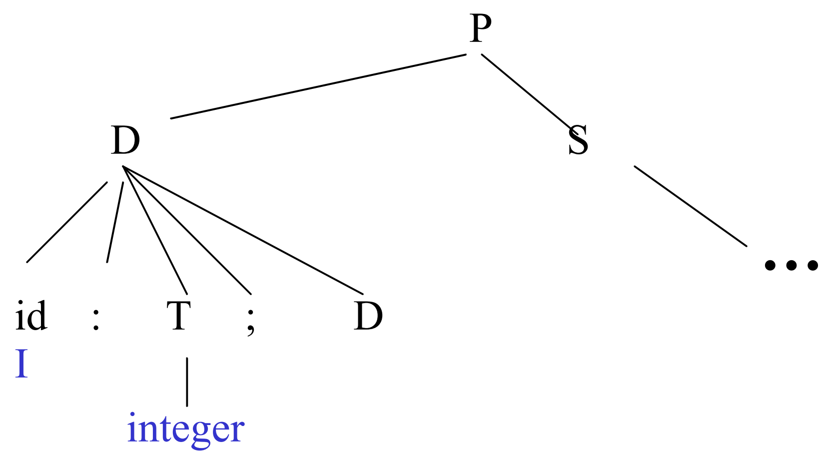
A: array[20] of integer;

B: array[20] of ^integer;

I := B[A[2]]^




Parse Tree



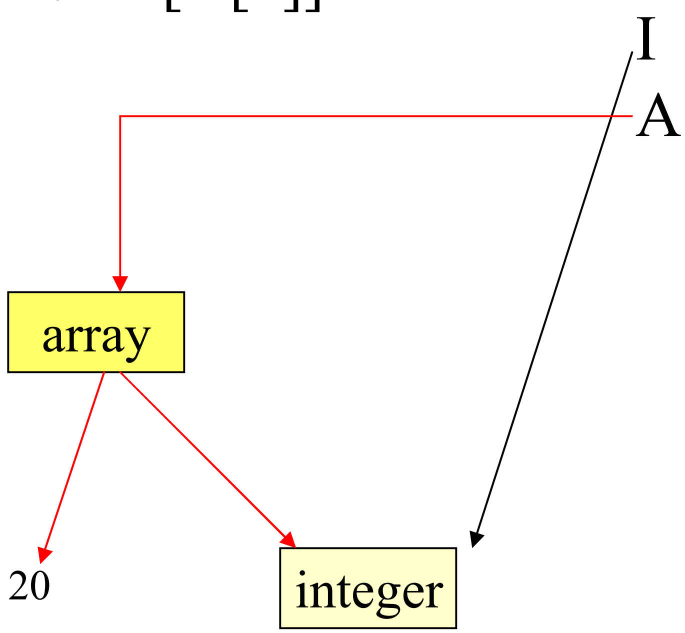
# Example

I: integer;

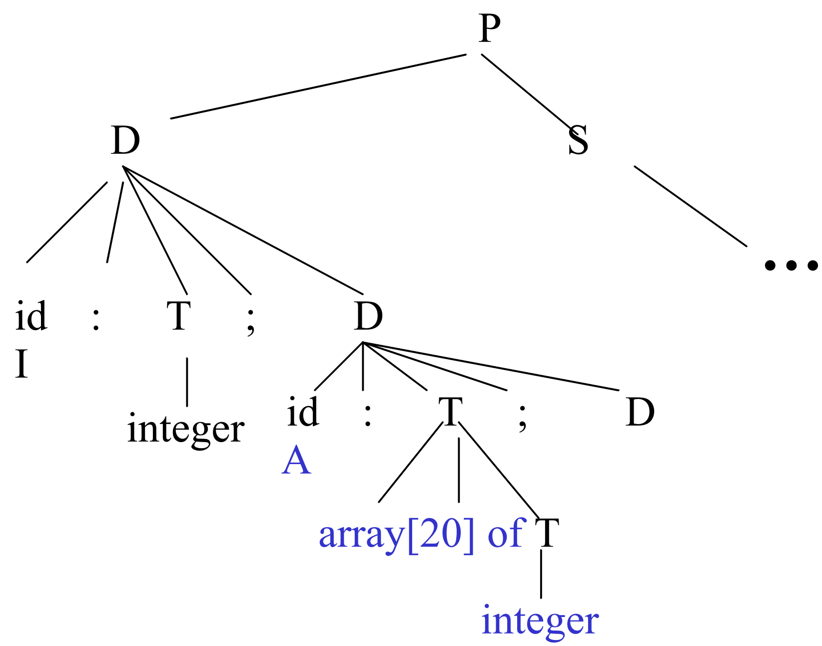
A: array[20] of integer;

B: array[20] of ^integer;

I := B[A[2]]^




Parse Tree



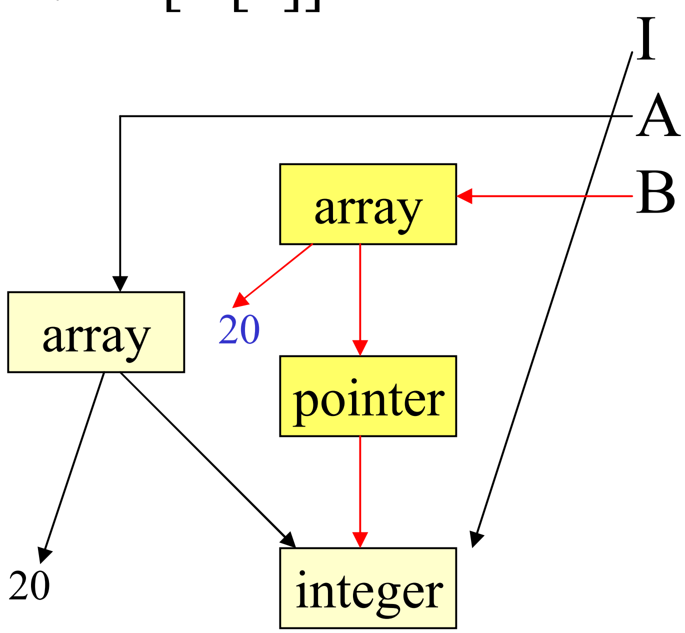
# Example

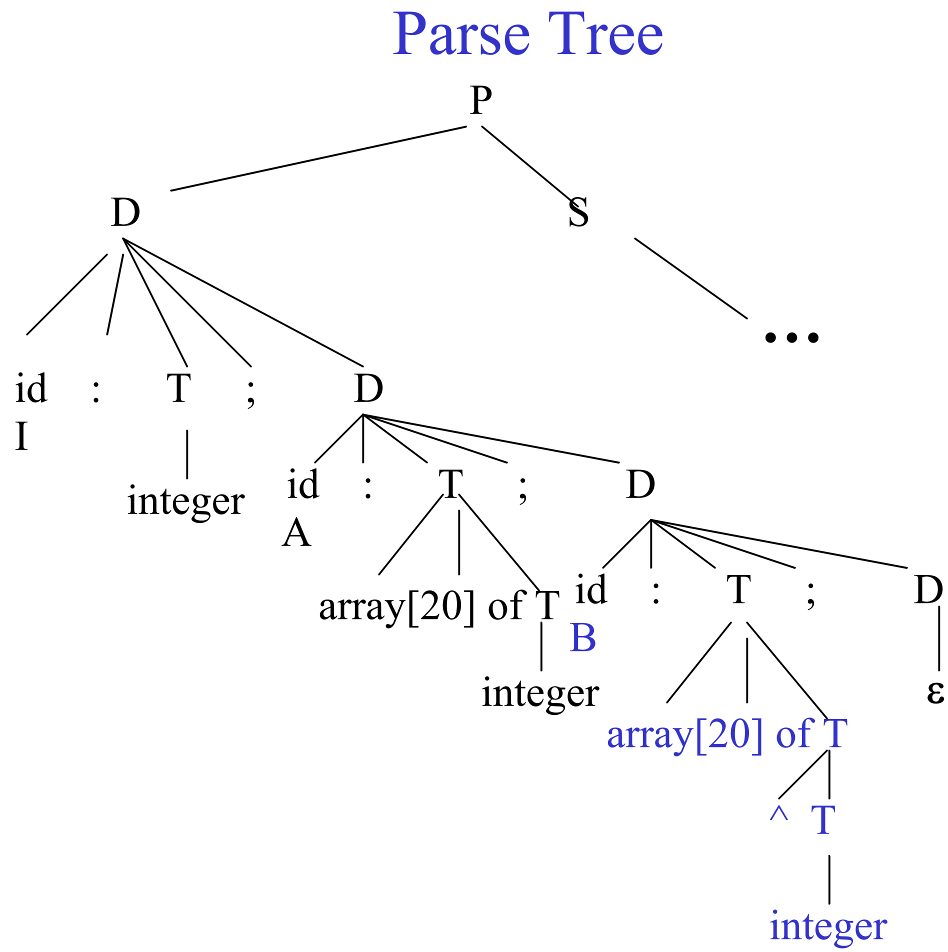
I: integer;

A: array[20] of integer;

B: array[20] of ^integer;

I := B[A[2]]^



# Typechecking Expressions

E → int_literal { E.type := integer; }
E → float_literal { E.type = float; }
E → id { E.type := lookup(id.name); }
E → $E_1 + E_2$ { if ($E_1$.type = integer & $E_2$.type = integer)
then E.type = integer;
else if ($E_1$.type = float & $E_2$.type = float)
then E.type = float;
else type_error(); }
E → $E_1$ [ $E_2$ ] { if ($E_1$.type = array of T & $E_2$.type = integer)
then E.type = T; else …}
E → $E_1$^ { if ($E_1$.type = ^T)
then E.type = T; else …}

These rules define a type system for the language

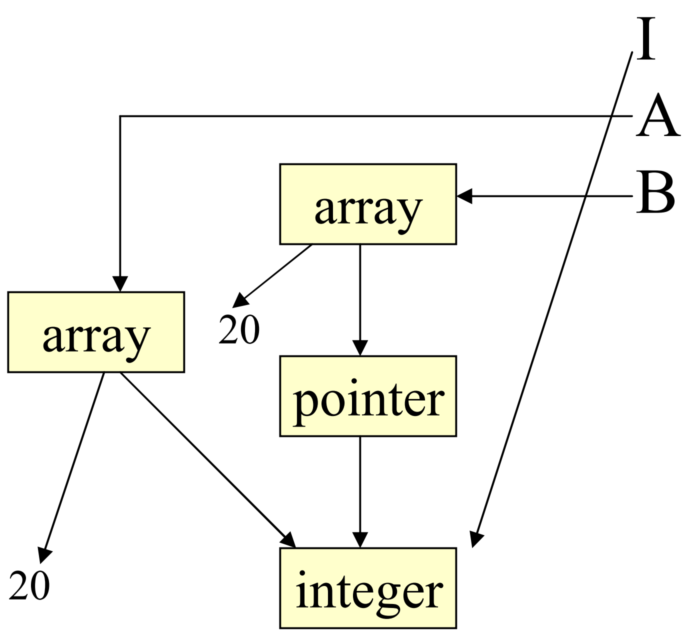
# Example

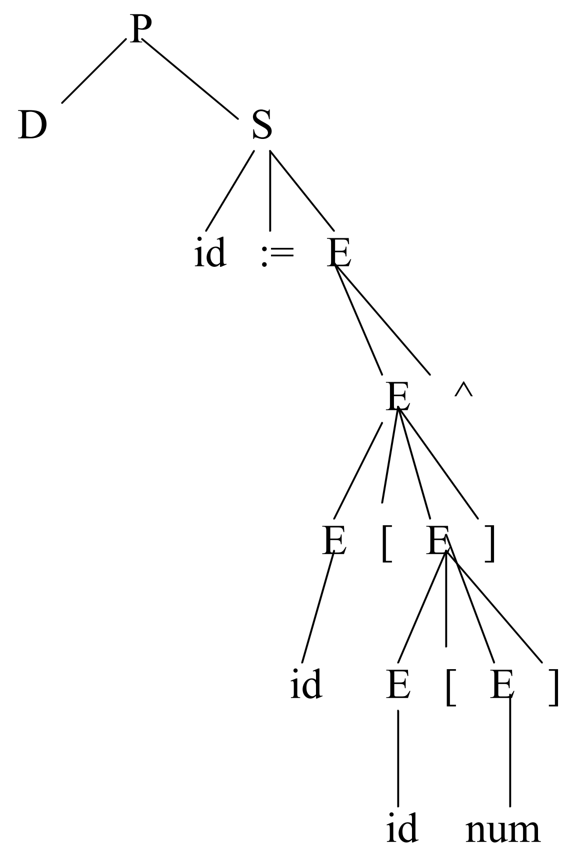
I: integer;

A: array[20] of integer;

B: array[20] of ^integer;

I := B[A[2]]^



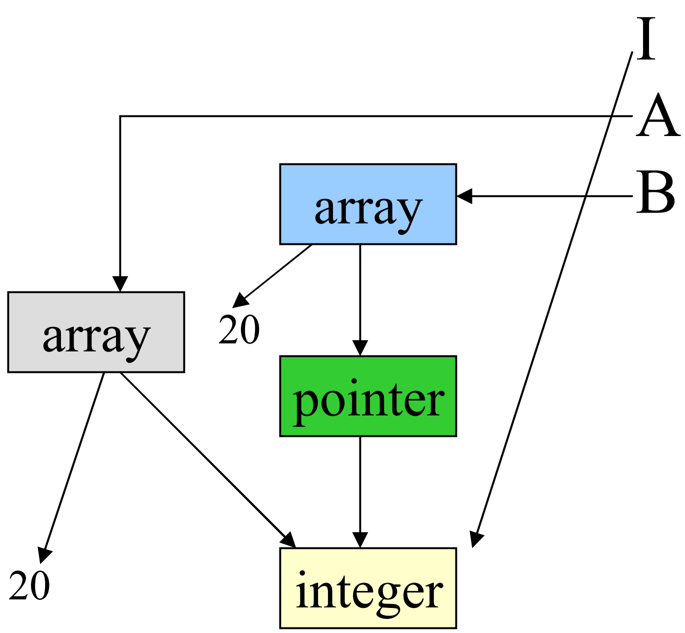
# Example

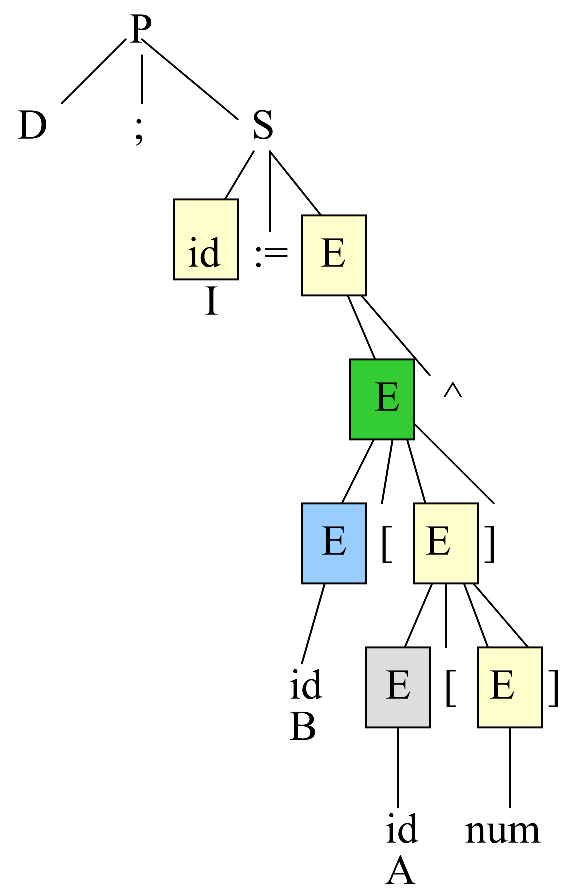
I: integer;

A: array[20] of integer;

B: array[20] of ^integer;

I := B[A[2]]^



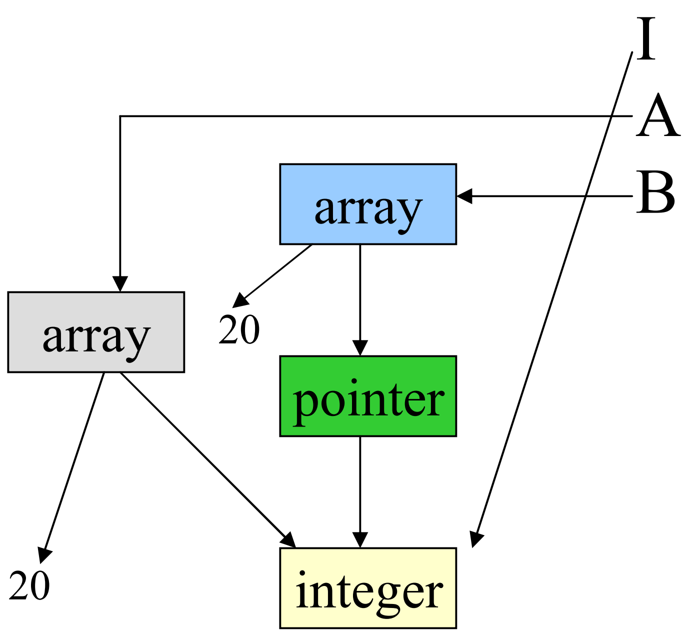
# Example

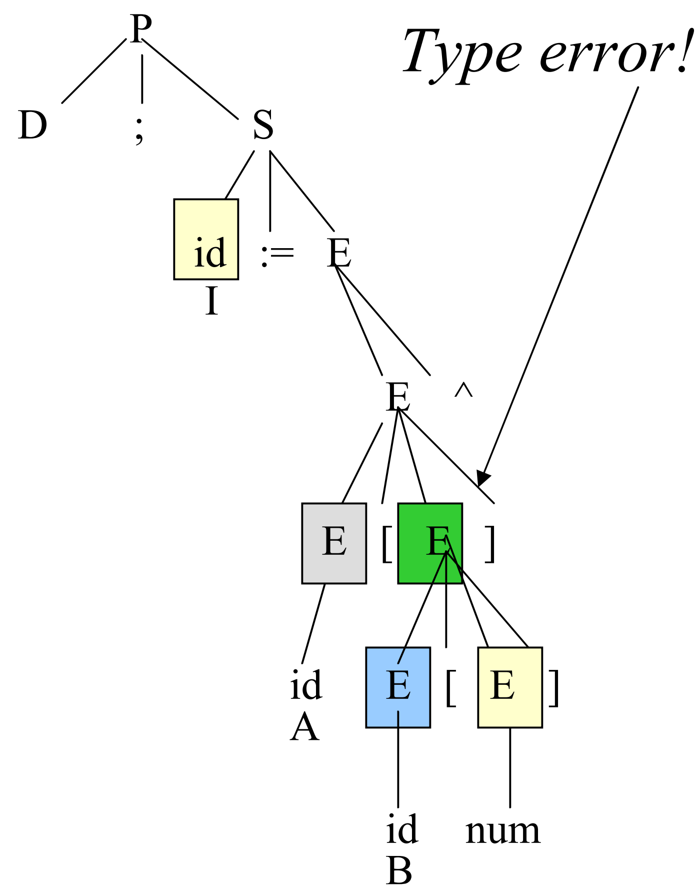
I: integer;

A: array[20] of integer;

B: array[20] of ^integer;

I := A[B[2]]^



# Typechecking Statements

$S \rightarrow S_1 ; S_1$ {if $S_1$.type = void & $S_1$.type = void)
then S.type = void; else error(); }

$S \rightarrow$ id := E { if lookup(id.name) = E.type
then S.type = void; else error(); }

$S \rightarrow$ if E then $S_1$ { if E.type = boolean and $S_1$.type = void
then S.type = void; else error();}

In this case, we assume that statements do not have types (not always the case).

# Typechecking Statements

What if statements have types?

$S \rightarrow S_1 ; S_2$ {S.type = $S_2$.type;}

$S \rightarrow$ id := E { if lookup(id.name) = E.type then
S.type = E.type; else error();
}

$S \rightarrow$ if E then $S_1$ else $S_2$
{ if (E.type = boolean & $S_1$.type = $S_2$.type) then
S.type = $S_1$.type;
else error();
}

# Untyped languages

Single type that contains all values

- Ex:
  - Lisp – program and data interchangeable
  - Assembly languages – bit strings
- Checking typically done at runtime

# Typed languages

- Variables have nontrivial types which limit the values that can be held.
- In most typed languages, new types can be defined using type operators.
- Much of the checking can be done at compile time!
- Different languages make different assumptions about type semantics.

# Components of a Type System

- Base Types
- Compound/Constructed Types
- Type Equivalence
- Inference Rules (Typechecking)
- …

Different languages make different choices!

# Base (built-in) types

- Numbers
  - Multiple – integer, floating point
  - precision
- Characters
- Booleans

# Constructed Types

- Array
- String
- Enumerated types
- Record
- Pointer
- Classes (OO) and inheritance relationships
- Procedure/Functions
- …

# Type Equivalence

Two types: Structural and Name

Type A = Bool

Type B = Bool

- In Structural equilivance:  Types A and B match because they are both boolean.
- In Name equilivance:  A and B don't match because they have different names.

# Implementing Structural Equivalence

To determine whether two types are structurally equilivant, traverse the types:

```
boolean equiv(s,t) {
    if s and t are same basic type return true
    if s = array(s1,s2) and t is array(t1,t2)
       return equiv(s1,t1) & equiv(s2,t2)
    if s = pointer(s1) and t = pointer(t1)
       return equiv(s1,t1)
   …
   return false;
}
```

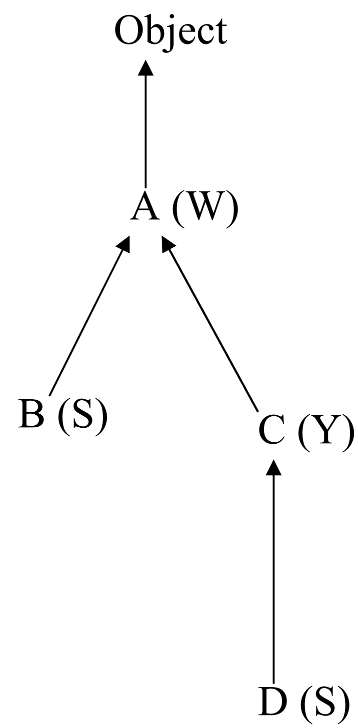
# Other Practical Type System Issues

- Implicit versus explicit type conversions
  - Explicit ➔ user indicates (Ada)
  - Implicit ➔ built-in (C int/char) -- coercions
- Overloading – meaning  based on context
  - Built-in
  - Extracting meaning – parameters/context
- Objects (inheritance)
- Polymorphism

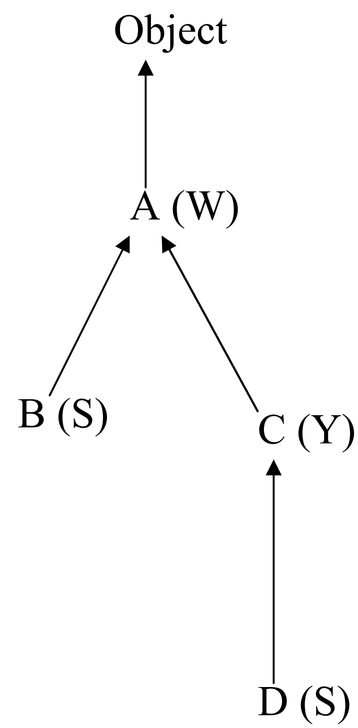
# OO Languages

- Data is organized into classes and sub-classes
- Top level is class of all objects
- Objects at any level inherit the attributes (data, functions) of objects higher up in the hierarchy. The subclass has a larger set of properties than the class. Subclasses can override behavior inherited from parent classes. (But cannot revise private data elements from a parent).

```
class A {
  public: A() {cout << "Creating A\n"; }
  W() {cout << "W in A\n"; }
};
class B: public A {
  public: B() {cout << "Creating B\n"; }
  S() {cout << "S in B\n"; }
};
class C: public A {
  public: C() {cout << "Creating C\n"; }
  Y() {cout << "Y in C\n"; }
};
class D: public C {
  public: D() {cout << "Creating D\n"; }
  S() {cout << "S in D\n"; }
};
```



| **The Code**: | **Output:** |
| --- | --- |
| `B b;` | Creating A<br>Creating B |
| `D d;` | Creating A<br>Creating C<br>Creating D |
| `b.W();` | W in A |
| `b.S();` | S in B |
| `d.W();` | W in A |
| `d.Y();` | Y in C |
| `d.S();` | S in D |



# OO Principle of Substitutability

- Subclasses possess all data areas associated with parent classes
- Subclasses implement (through inheritance) at least all functionality defined for the parent class

**If we have two classes, A and B, such that class B is a subclass of A (perhaps several times removed), it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect.**

# Typechecking OO languages

- Without inheritance, the task would be relatively simple (similar to records)
- Difficulties:
  - Method overriding
  - When can super/sub types be used? Consider function f: A → B
    - Actual parameter of type A or subtype of A
    - Return type B or supertype of B
  - Multiple inheritance

# Function parameters

- Function parameters make typechecking more difficult

```
procedure mlist(lptr: link; procedure p)
  while lptr <> nil begin
      p(lptr);
      lptr = lptr→next;
   end
end
```

# Polymorphism

- Functions – statements in body can be executed on arguments with different type – common in OO languages because of inheritance
- Ex: Python for determining the length of a list

```
def size (lis):
    if null(lis):
         return 0
    else:
         return size(lis[1:]) + 1;

size([‘sun’,’mon’,’tue’])
size([10,11,12])
size(A)
```

# Type Inferencing

```
def size (lis):
  if null(lis):
    return 0
  else:
    return size(lis[1:])+1;
```

| Expression | Type |
|---|---|
| **size** | $\beta \rightarrow \gamma$ |
| **lis** | $\beta$ |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

**Goal: determine a type for `size` so we can typecheck the calls.**

**Greek symbols are *type variables*.**

Fig 6.30 of your text

# Type Inferencing

```
def size (lis):
  if null(lis):
    return 0
  else:
    return size(lis[1:])+1;
```

**Built-in language constructs and functions provide clues.**

**Given what we have in the table, we now know that** $list(\alpha_n) = \beta$

Fig 6.30 of your text

| Expression | Type |
|---|---|
| **size** | $\beta \rightarrow \gamma$ |
| **lis** | $\beta$ $(list(\alpha_n))$ |
| **if** | $bool \times \alpha_i \times \alpha_i \rightarrow \alpha_i$ |
| **null** | $list(\alpha_n) \rightarrow bool$ |
| **null(lis)** | $bool$ |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

# Type Inferencing

```
def size (lis):
  if null(lis):
    return 0
  else:
    return size(lis[1:])+1;
```

$\alpha_i$ **= int**

| Expression | Type |
|---|---|
| **size** | $\beta \rightarrow \gamma$ |
| **lis** | $\beta$ (*list*($\alpha_n$) ) |
| **if** | *bool* x $\alpha_i$ x $\alpha_i \rightarrow \alpha_i$ |
| **null** | *list*($\alpha_n$) $\rightarrow$ *bool* |
| **null(lis)** | *bool* |
| **0** | *int* |
| **+** | *int* x *int* $\rightarrow$ *int* |
| **lis[1:]** | *list*($\alpha_n$) |
| | |
| | |
| | |
| | |

Fig 6.30 of your text

# Type Inferencing

```
def size (lis):
  if null(lis):
    return 0
  else:
    return size(lis[1:])+1;
```

$\gamma = int$

| Expression | Type |
|---|---|
| **`size`** | $\beta \rightarrow \gamma$ |
| **`lis`** | $\beta$ ($list(\alpha_n)$ ) |
| **if** | $bool \times \alpha_i \times \alpha_i \rightarrow \alpha_i$ |
| **`null`** | $list(\alpha_n) \rightarrow bool$ |
| **`null(lis)`** | *bool* |
| **`0,1`** | *int* |
| **`+`** | $int \times int \rightarrow int$ |
| **`lis[1:]`** | $list(\alpha_n)$ |
| **`size(lis[1:])`** | $\gamma$ |
| **`size(lis[1:]) + 1`** | *int* |
| **`if(…)`** | *int* |

**All of this tells us that**
**`size`: $list(\alpha) \rightarrow int$**
**(in other words, maps from anything with type list to type integer)**

Fig 6.30 of your text

# Formalizing Type Systems

- Mathematical characterizations of the type system – Type soundness theorems.
- Requires formalization of language syntax, static scoping rules and semantics.
- Formalization of type rules

- http://research.microsoft.com/users/luca/Papers/TypeSystems.pdf

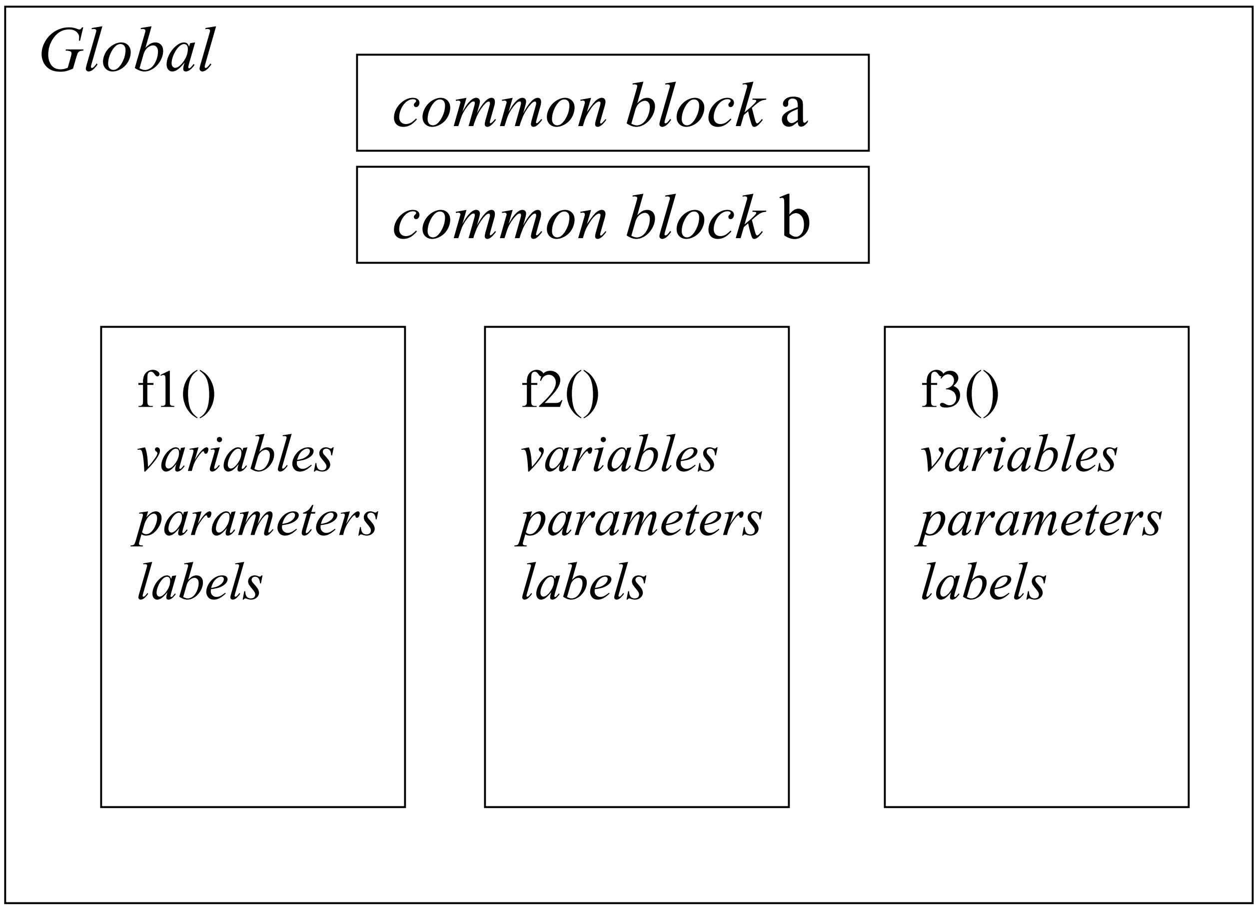
# Part II: Scope

# Scope

In most languages, a complete program will contain several different **namespaces** or **scopes**.

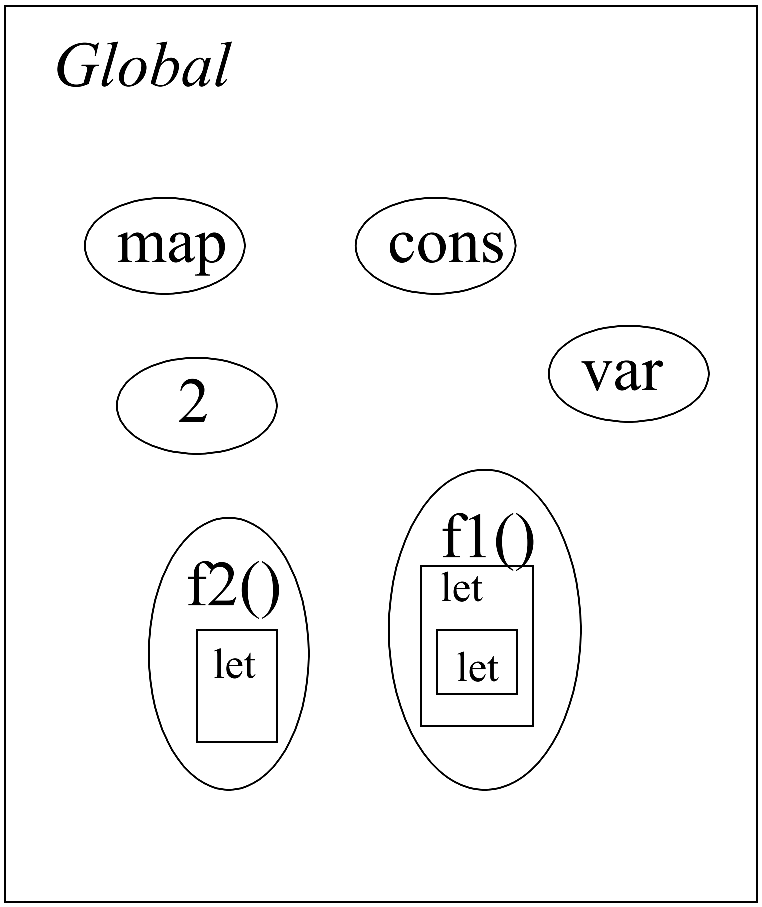
Different languages have different rules for namespace definition

# Fortran 77 Name Space



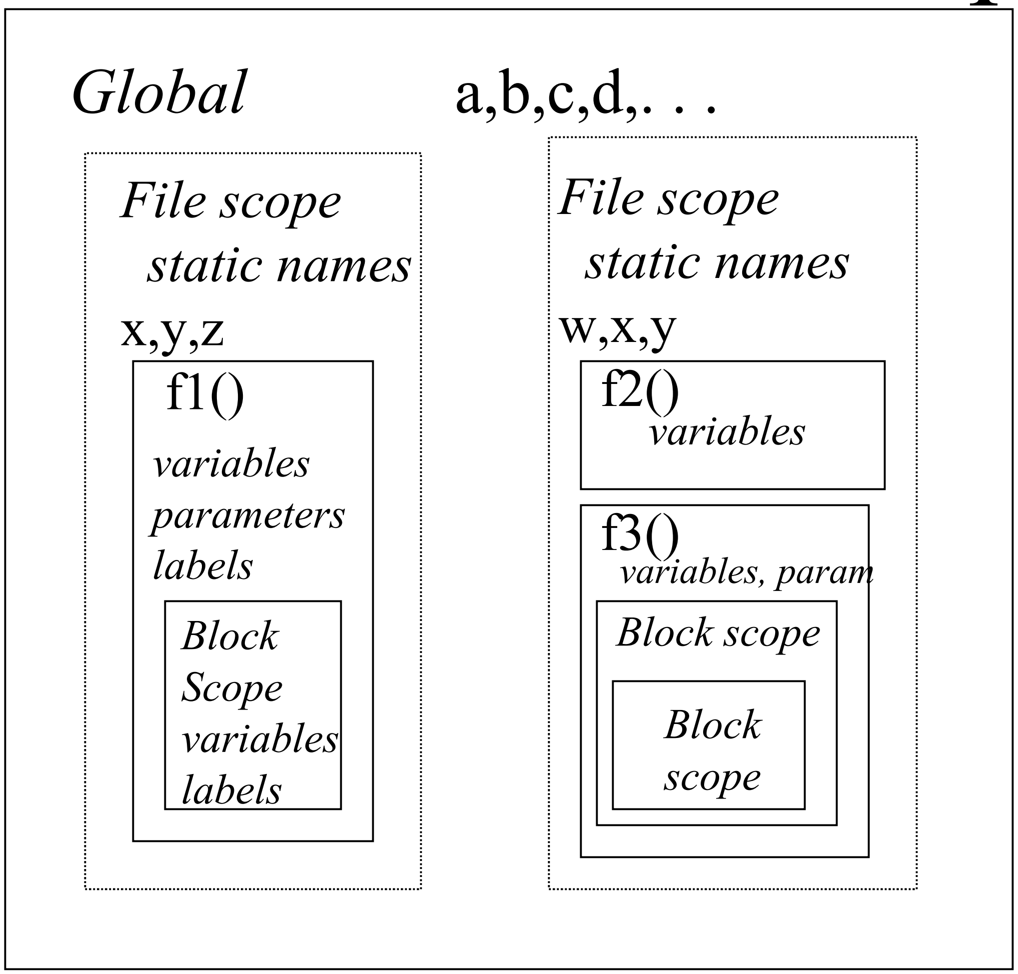

Global scope holds procedure names and common block names. Procedures have local variables parameters, labels and can import common blocks

# Scheme Name Space



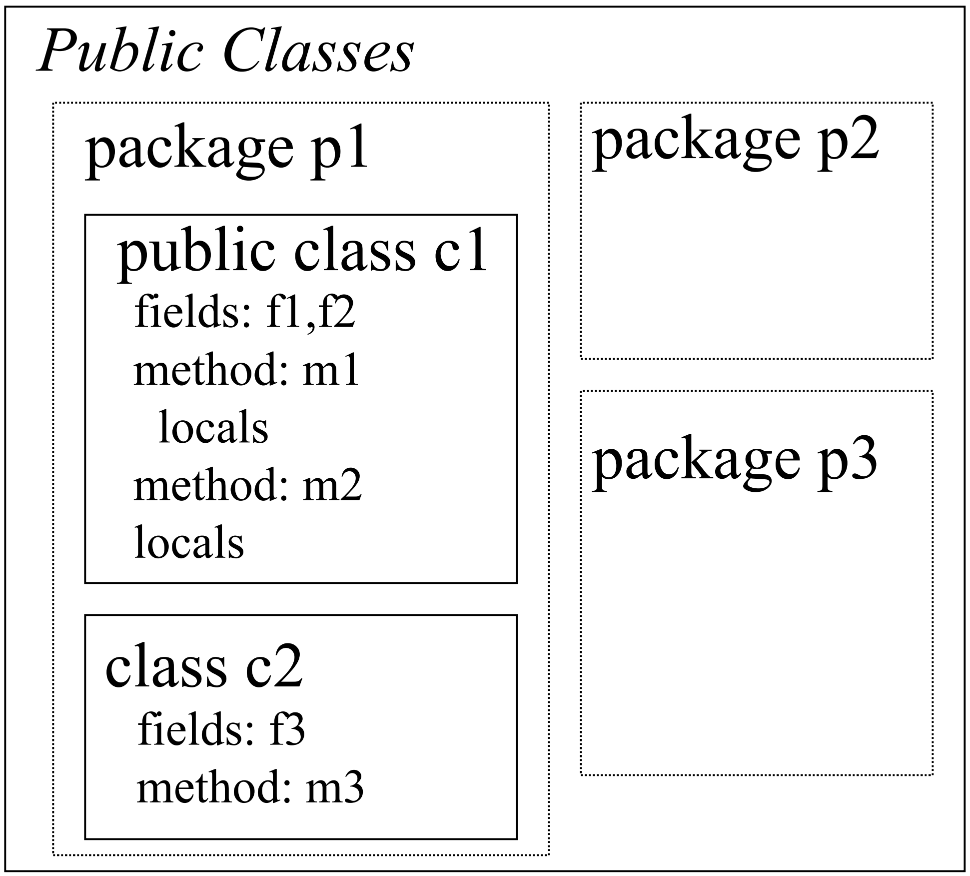

- All objects (built-in and user-defined) reside in single global namespace
- 'let' expressions create nested lexical scopes

# C Name Space




- Global scope holds variables and functions
- No function nesting
- Block level scope introduces variables and labels
- File level scope with static variables that are not visible outside the file (global otherwise)

# Java Name Space




- Limited global name space with only public classes
- Fields and methods in a public class can be public ➔ visible to classes in other packages
- Fields and methods in a class are visible to all classes in the same package unless declared private
- Class variables visible to all objects of the same class.

# Scope

Each **scope** maps a set of variables to a set of meanings.

The **scope of a variable declaration** is the part of the program where that variable is visible.

# Referencing Environment

The **referencing environment** at a particular location in source code is the set of variables that are visible at that point.

- A variable is **local** to a procedure if the declaration occurs in that procedure.
- A variable is **non-local** to a procedure if it is visible inside the procedure but is not declared inside that procedure.
- A variable is **global** if it occurs in the outermost scope (special case of non-local).

# Types of Scoping

- Static – scope of a variable determined from the source code.
  - "Most Closely Nested"
  - Used by most languages
- Dynamic – current call tree determines the relevant declaration of a variable use.

# Static: Most Closely Nested Rule

The scope of a particular declaration is given by the most closely nested rule

- The scope of a variable declared in block B, includes B.
- If x is not declared in block B, then an occurrence of x in B is in the scope of a declaration of x in some enclosing block A, such that A has a declaration of x and A is more closely nested around B than any other block with a declaration of x.

# Example Program: Static

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```

What is visible at this point (globally)?

# Example Program: Static

```
Program main;
   a,b,c: real;
   procedure sub1(a: real);
      d: int;
      procedure sub2(c: int);
            d: real;
      body of sub2
      procedure sub3(a:int)
         body of sub3
   body of sub1
body of main
```

What is visible at this point (sub1)?

# Example Program: Static

```
Program main;
   a,b,c: real;
   procedure sub1(a: real);
      d: int;
      procedure sub2(c: int);
            d: real;
      body of sub2
      procedure sub3(a:int)
         body of sub3
   body of sub1
body of main
```

What is visible at this point (sub3)?

# Example Program: Static

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
         d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```

What is visible at this point (sub2)?

# Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching the chain of subprogram calls (runtime stack) that forced execution to this point

# Scope Example

```
MAIN
     - declaration of x
        SUB1
          - declaration of x -
          ...
          call SUB2
          ...

        SUB2
          ...
          - reference to x -
          ...

     ...
     call SUB1
     …
```

**MAIN calls SUB1**
**SUB1 calls SUB2**
**SUB2 uses x**

Which x??

# Scope Example

```
MAIN
   - declaration of x
      SUB1
        - declaration of x -
        ...
        call SUB2
        ...

      SUB2
       ...
       - reference to x -
       ...

   ...
   call SUB1
   …
```

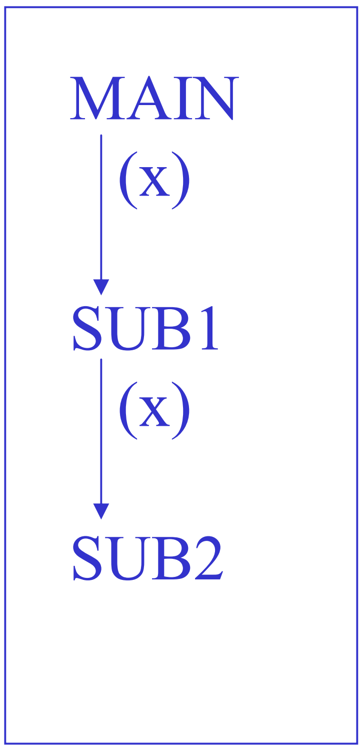
**MAIN calls SUB1**
**SUB1 calls SUB2**
**SUB2 uses x**

For static scoping, it is main's x

# Scope Example

- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms.
- A subprogram is active if its execution has begun but has not yet terminated.

# Scope Example




```
MAIN
  - declaration of x
    SUB1
      - declaration of x -
      ...
      call SUB2
      ...

    SUB2
      ...
      - reference to x -
      ...

  ...
  call SUB1
  …
```

**MAIN calls SUB1**
**SUB1 calls SUB2**
**SUB2 uses x**

For dynamic scoping, it is sub1's x

# Dynamic Scoping

- Evaluation of Dynamic Scoping:
  - Advantage: convenience (easy to implement)
  - Disadvantage: poor readability, unbounded search time

# Part III: Symbol Tables

# Symbol Table

- Primary data structure inside a compiler.
- Stores information about the symbols in the input program including:
  - Type (or class)
  - Size (if not implied by type)
  - Scope
- Scope represented explicitly or implicitly (based on table structure).
- Classes can also be represented by structure – one difference = information about classes must persist after have left scope.
- Used in all phases of the compiler.

# Symbol Table Object

Symbol table functions are called during parsing:

- Insert(x) –*A new symbol is defined*.
- Delete(x) –*The lifetime of a symbol ends*.
- Lookup(x) –*A symbol is used*.
- EnterScope(s) – *A new scope is entered*.
- ExitScope(s) – *A scope is left*.

# Scope and Parsing

```
func_decl   : FUNCTION NAME                       {EnterScope($2);}
                parameter decls stmts ;           {ExitScope($2); }

decl        :     name ':'   type                 {Insert($1,$3); }

…
statements:   id := expression                    {lookup($1);}
…
expression:        …
              id                                  {lookup($1);}
```

*Note: This is a greatly simplified grammar including only the symbol table relevant productions.*

# Symbol Table Implementation

- Variety of choices, including arrays, lists, trees, heaps, hash tables, …
- Different structures may be used for local tables versus tables representing scope.

# Example Implementation

- Local level – within a scope, use a table or linked list.
- Global – each scope is represented as a structure that points at –
  - Its local symbols
  - The scopes that it encloses
  - Its enclosing scope

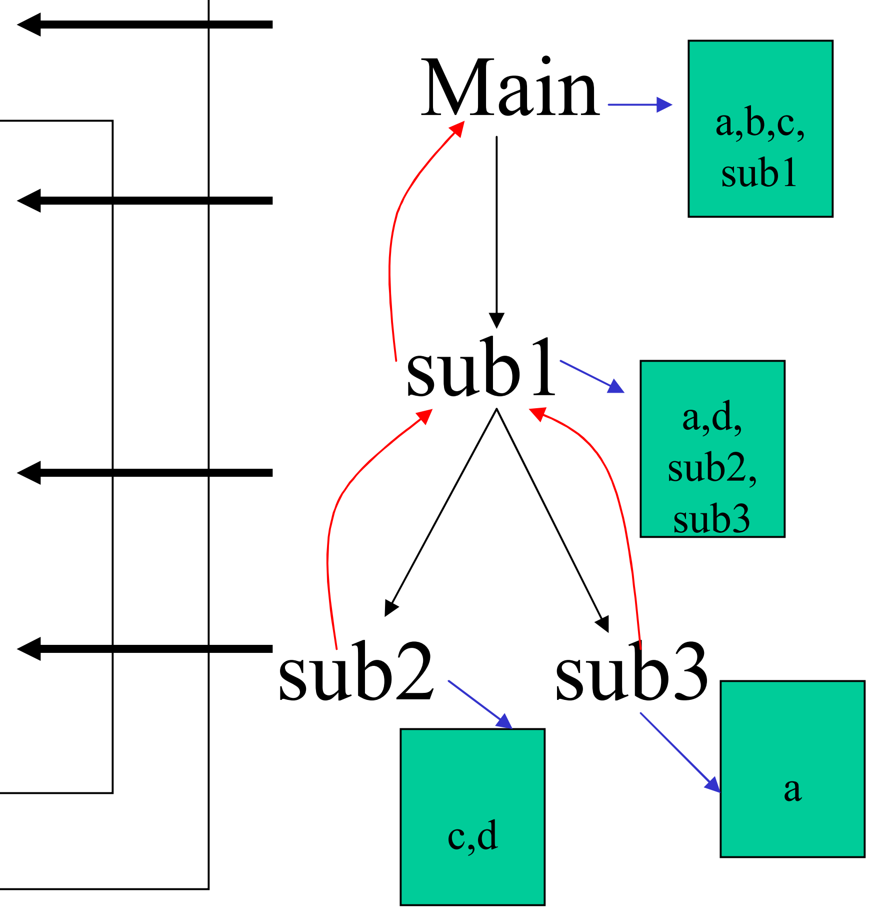
} → A tree?

# Implementing the table

- Need variable CS for current scope
- *EnterScope* – creates a new record that is a child of the current scope. This scope has new empty local table. Set CS to this record.
- *ExitScope* – set CS to parent of current scope. Update tables.
- *Insert* – add a new entry to the local table of CS
- *Lookup* – Search local table of CS. If not found, check the enclosing scope. Continue checking enclosing scopes until found or until run out of scopes.

# Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



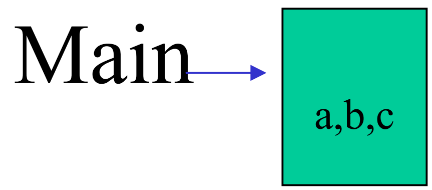
# Implementing the table

We can use a stack instead!!!

- *EnterScope* – creates a new record that is a child of the current scope. This scope has new empty local table. Set CS to this record ➔ PUSH
- *ExitScope* – set CS to parent of current scope. Update tables ➔ POP
- *Insert* – add a new entry to the local table of CS
- *Lookup* – Search local table of CS. If not found, check the enclosing scope. Continue checking enclosing scopes until found or until run out of scopes.

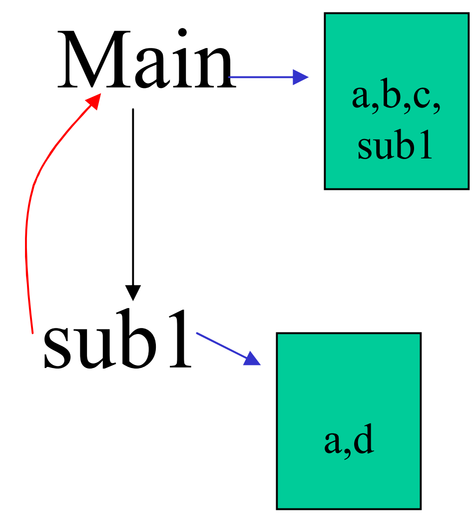
# Example Program – As we compile …

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



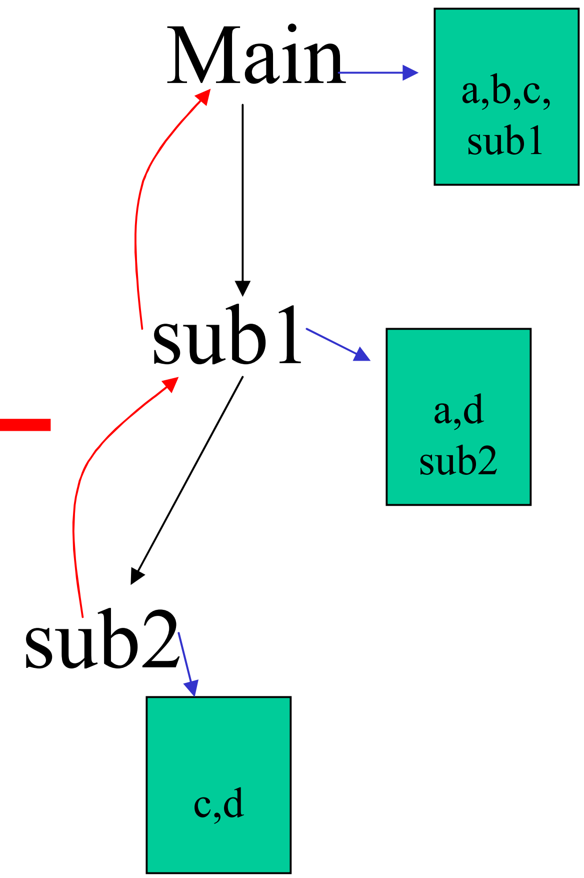
# Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



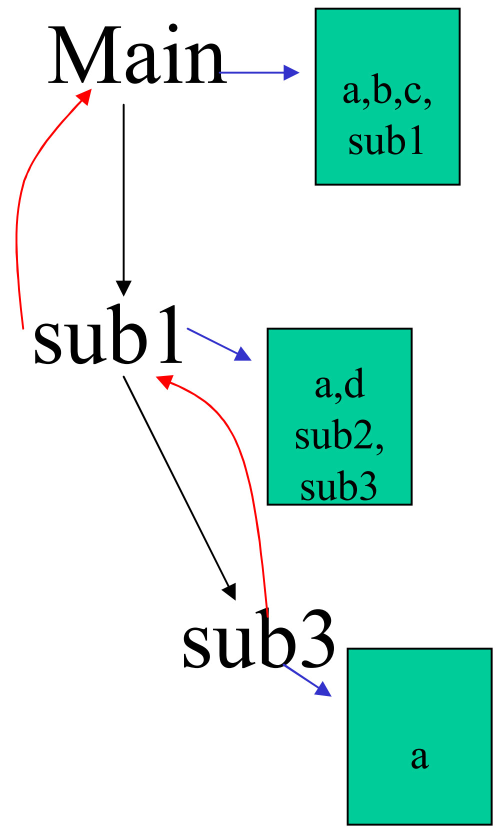
# Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



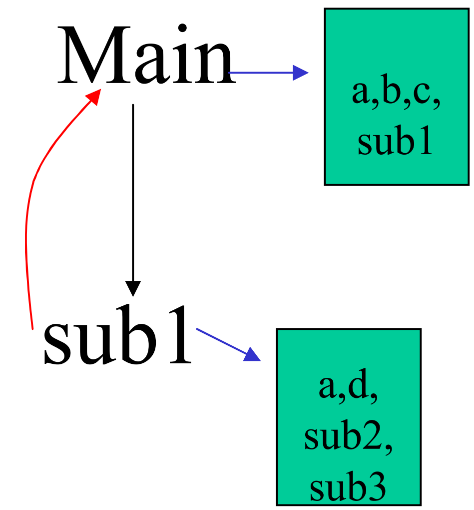
# Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



# Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



# Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```