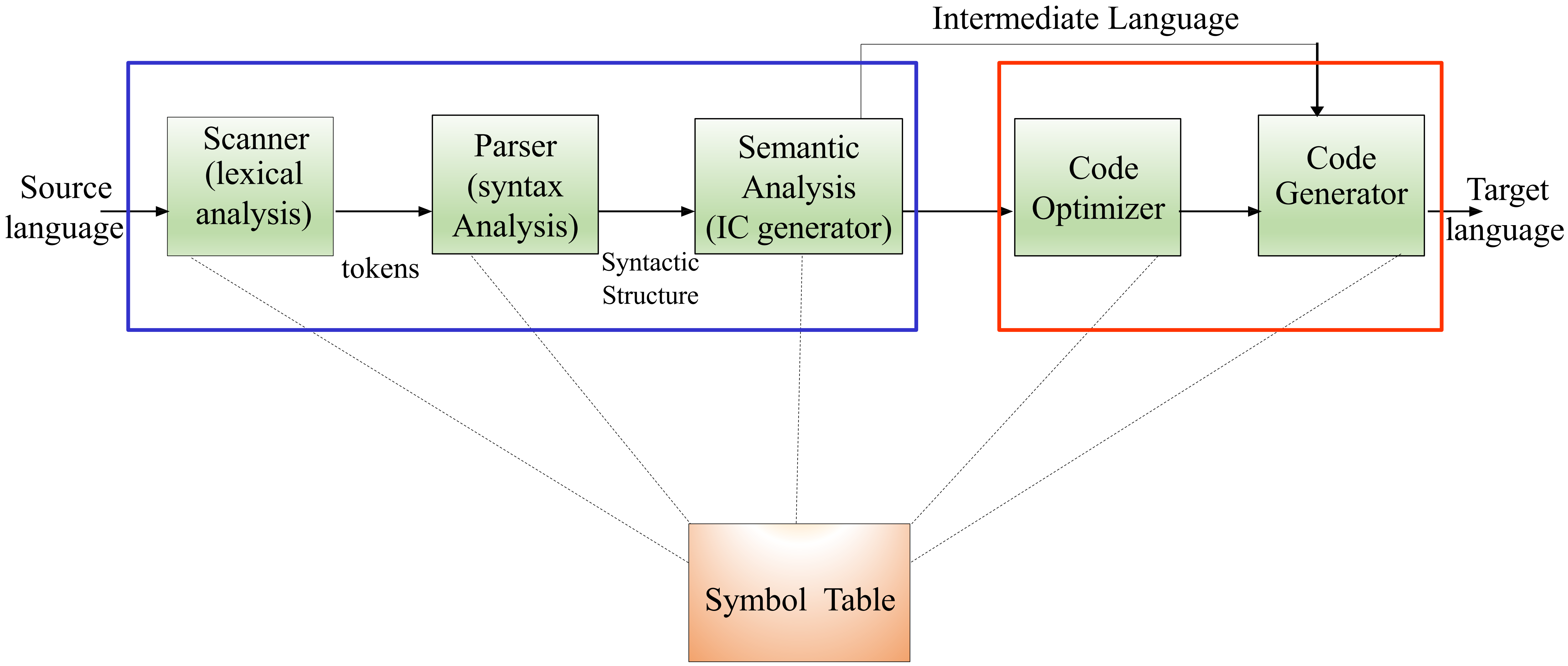
CSC 340

Semantic Analysis

Compiler Architecture



Semantics

- Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other.
- Semantic analysis judges whether the syntax structure constructed in the source program derives <u>any meaning or not.</u>
- CFG + semantic rules = Syntax Directed Definitions
- For example:
  - int a = “value”;
  - should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs.
  - These rules are set by the grammar of the language and evaluated in semantic analysis.

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 <u>type expressions</u> 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.

Type Expressions

- Types have structure, which we shall represent using type expressions:
  - a type expression is either a basic type
  - or is formed by applying an operator called a type constructor to a type expression.
- A basic type is a type expression. Typical basic types for a language include boolean, char, integer, float, and void; the latter denotes "the absence of a value."
- A type name is a type expression.
- A type expression can be formed by applying the array type constructor to a number and a type expression.

Type Expressions

- A record is a data structure with named fields. A type expression can be formed by applying the record type constructor to the field names and their types.
- A type expression can be formed by using the type constructor (→)for function types. We write s →t for "function from type s to type t."
- If s and t are type expressions, then their Cartesian product s x t is a type expression.

Type Expressions

- Examples of Type Expressions
- float xform[3][3];
  - xform ∈ array(array(float))
- char *string;
  - string ∈ pointer(char)
- struct list { int element; struct list *next; } l;
  - list ≡ record((element, int), (next, pointer(list))) l ∈ list
- int max(int, int);
- max ∈ →(tuple(int, int), int)

Inference Rules - Typechecking

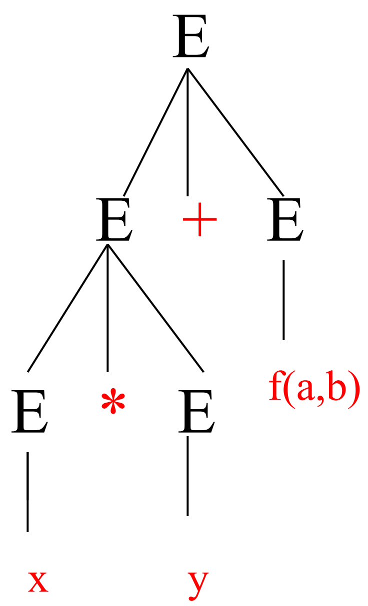
- **Static** (compile time) and **Dynamic** (runtime).
- One responsibility of a compiler is to see that all symbols are used correctly (i.e. consistently with the type system) to prevent *execution errors*.
- **Strong typing** – All expressions are guaranteed to be type consistent although the type 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, we need to:

1. **Process declarations** for given symbols
   - Scope – what are the visible symbols in the current scope?
   - Type – what is the declared type of the symbol?
2. **Lookup symbols** used in program to find current binding
3. **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

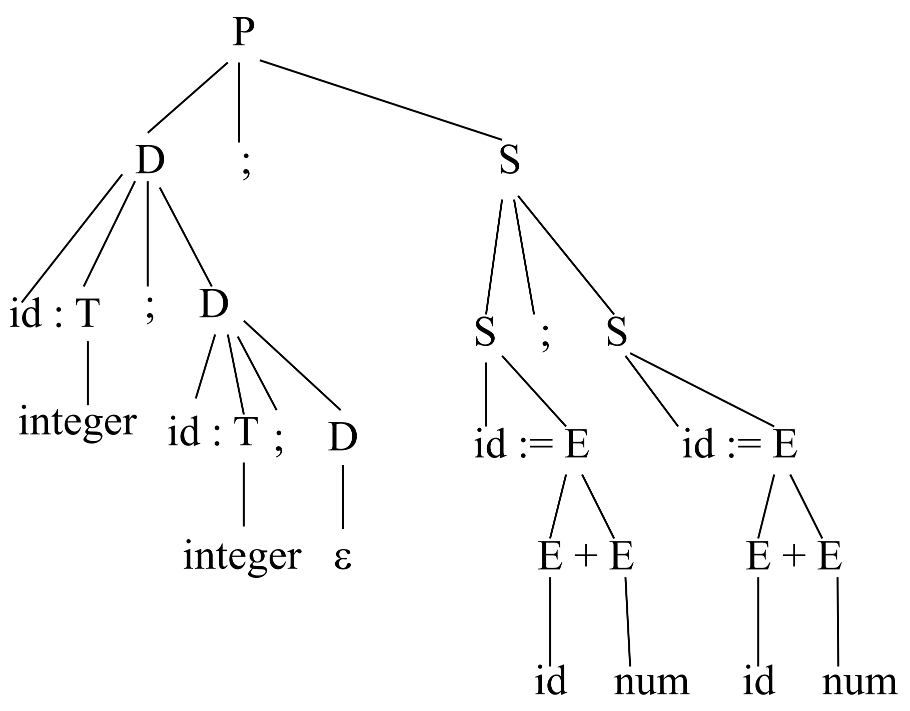
E → int_literal | float_literal | id | E + E | E [E] | E^

How can we typecheck strings in this language?

Example of language:

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^

```
i: integer; j: integer;
i := i + 1;
j := i + 1
```



Processing Declarations

D → id : T ; D {insert(id.name,T.type);}

Put info into the symbol table

D → ε

T → integer {T.type = integer;}

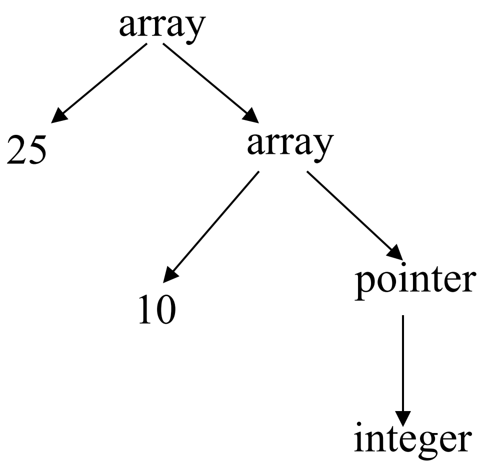
T → float {T.type = float;}

T → array [num] of T_1 {T.type = array(T_1.type,num); }

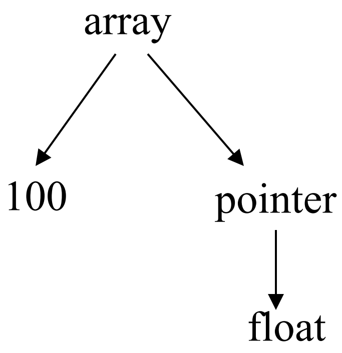
T → ^T_1 {T.type = pointer(T_1.type);}

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

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^

Example

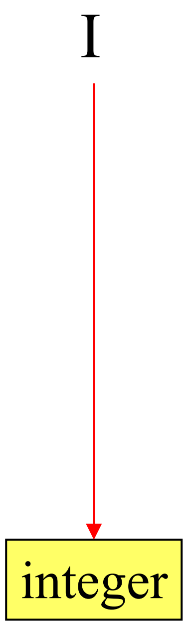
I: integer;

A: array[20] of integer;

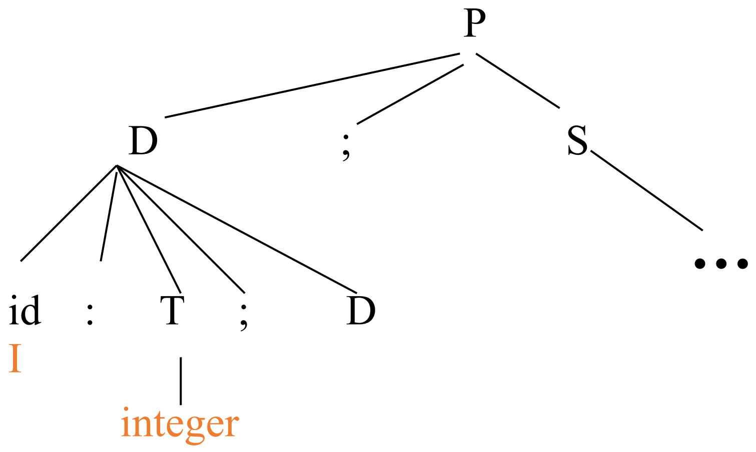
B: array[20] of ^integer;

I := B[A[2]]^

DAG



Parse Tree



Example

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^

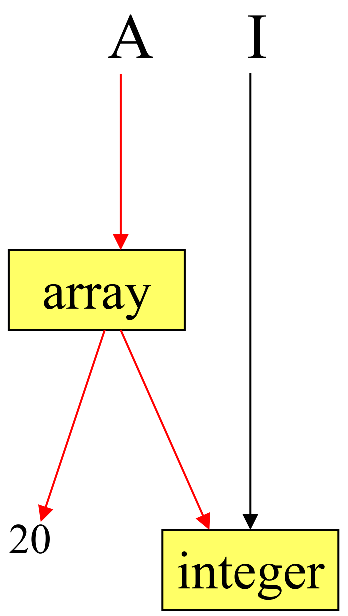
I: integer;

A: array[20] of integer;

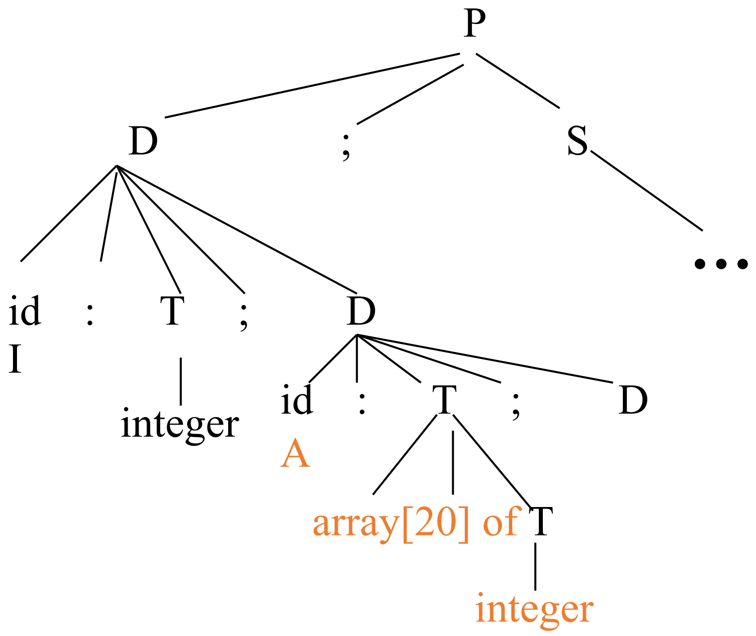
B: array[20] of ^integer;

I := B[A[2]]^

DAG



Parse Tree



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^

Example

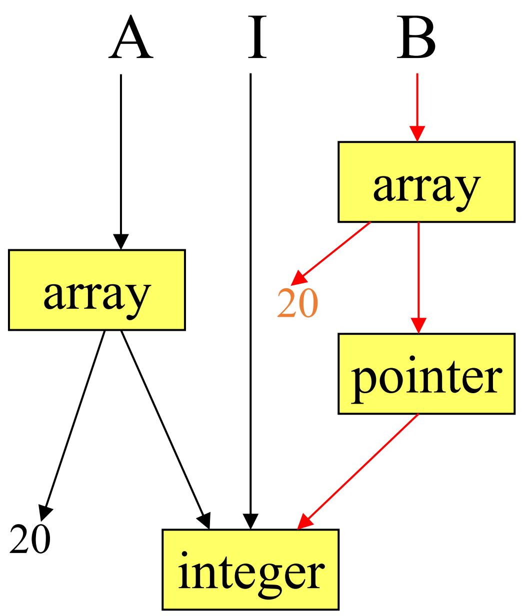
I: integer;

A: array[20] of integer;

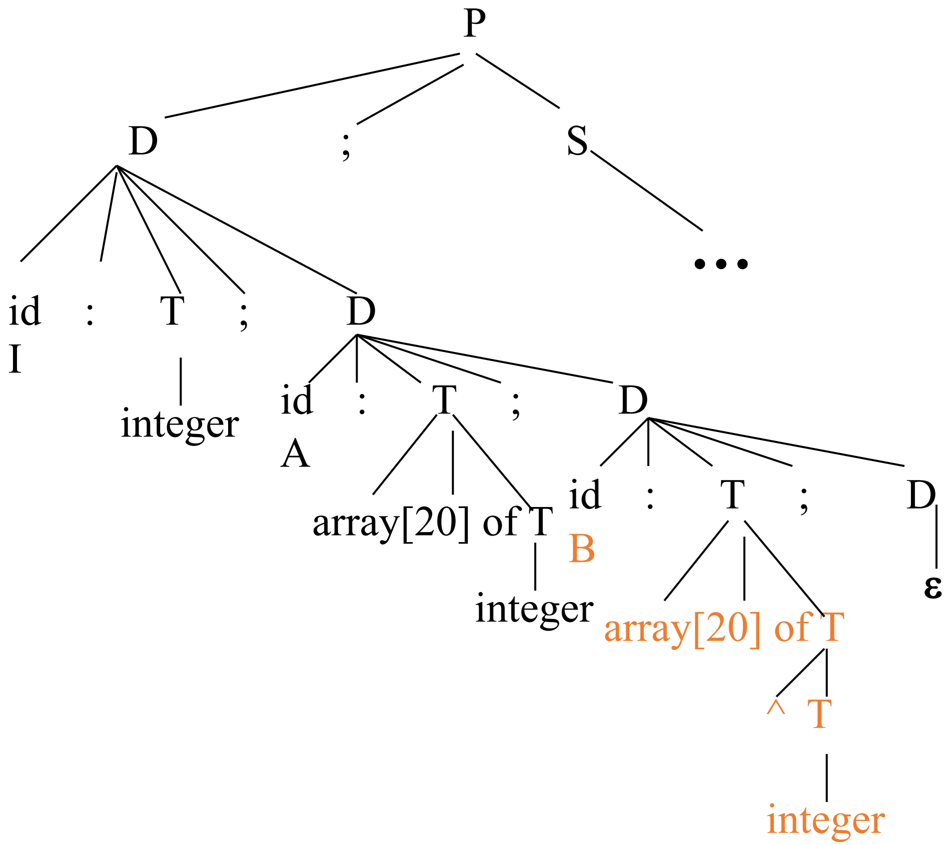
B: array[20] of ^integer;

I := B[A[2]]^

DAG



Parse Tree



A convenient way to represent a type expression is to use a graph.
construct a dag for a type expression, with interior nodes for type constructors and leaves for basic types

Typechecking Expressions

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^

```
E → int_literal   { E.type := integer; }
E → float_literal { E.type := float; }
E → id            { E.type := lookup(id.name); } //lookup from symbol table
E → E1 + E2       { if (E1.type = integer & E2.type = integer)
                     then  E.type = integer;
                     else if (E1.type = float & E2.type = float)
                     then  E.type = float;
                     else type_error(); }
E → E1 [ E2 ]     { if (E1.type = array of T & E2.type = integer)
                     then  E.type = T;
                     else …}
E → E1^           { if (E1.type = ^T)
                     then  E.type = T;
                     else …}
```

These rules (if-else) define
a type system for the language

Example

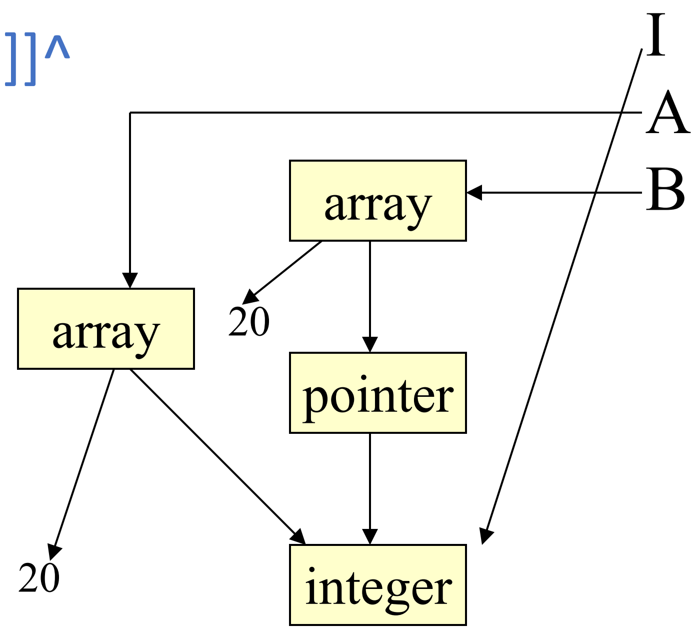
P → D ; S
D → id: T ; D | ε
T → integer | float | array [num] of T | ^T
S → S ; S | id := E
E → int_literal | float_literal | id | E + E | E [E] | E^

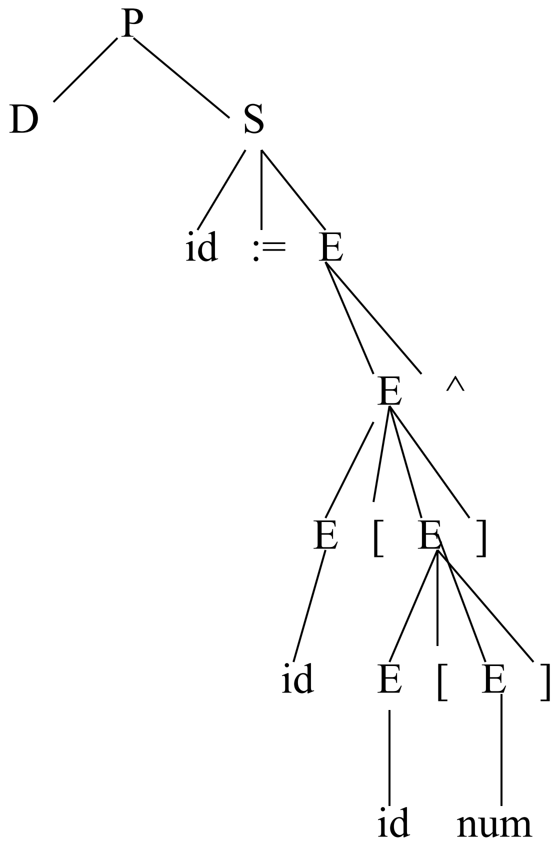
I: integer;

A: array[20] of integer;

B: array[20] of ^integer;

I := B[A[2]]^





Example

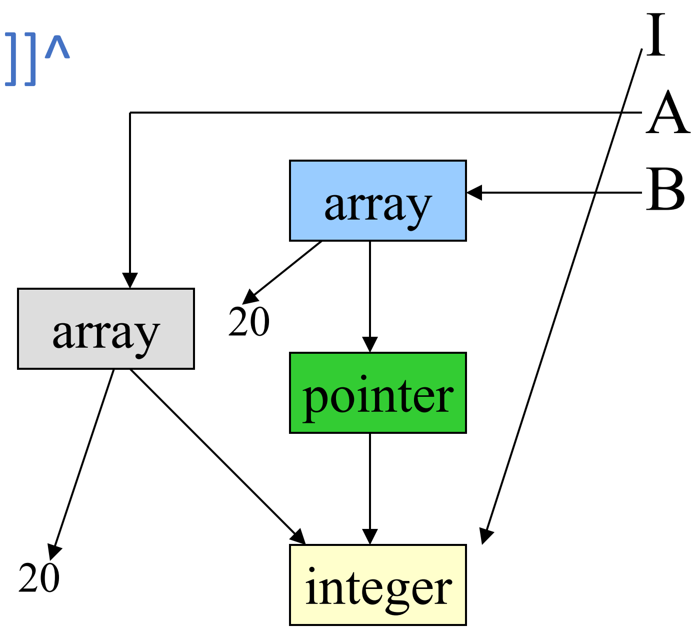
P → D ; S
D → id: T ; D | ε
T → integer | float | array [num] of T | ^T
S → S ; S | id := E
E → int_literal | float_literal | id | E + E | E [E] | E^

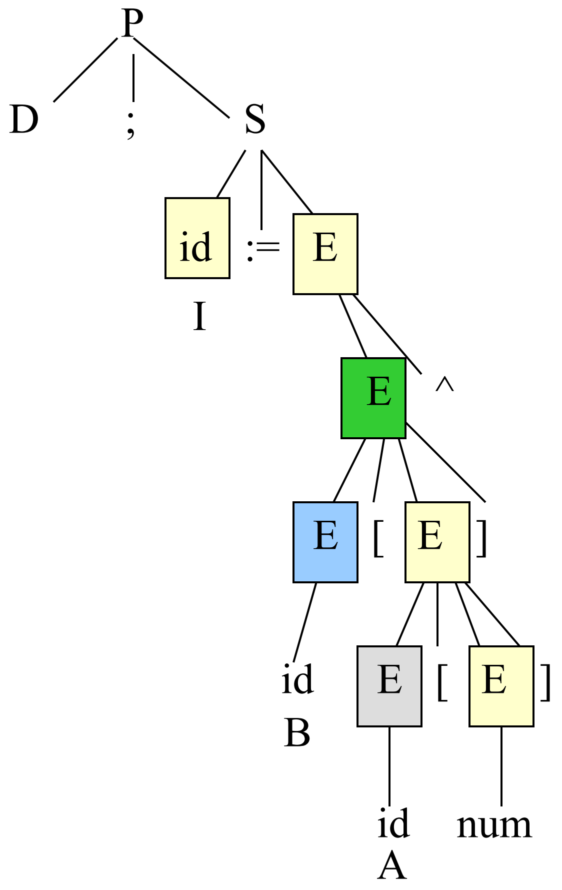
I: integer;

A: array[20] of integer;

B: array[20] of ^integer;

I := B[A[2]]^





Example

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^

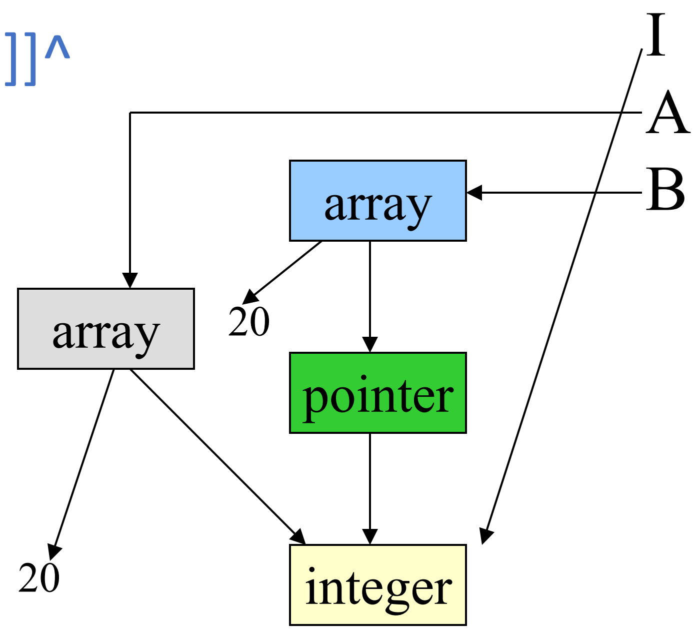
What if we switch A and B?

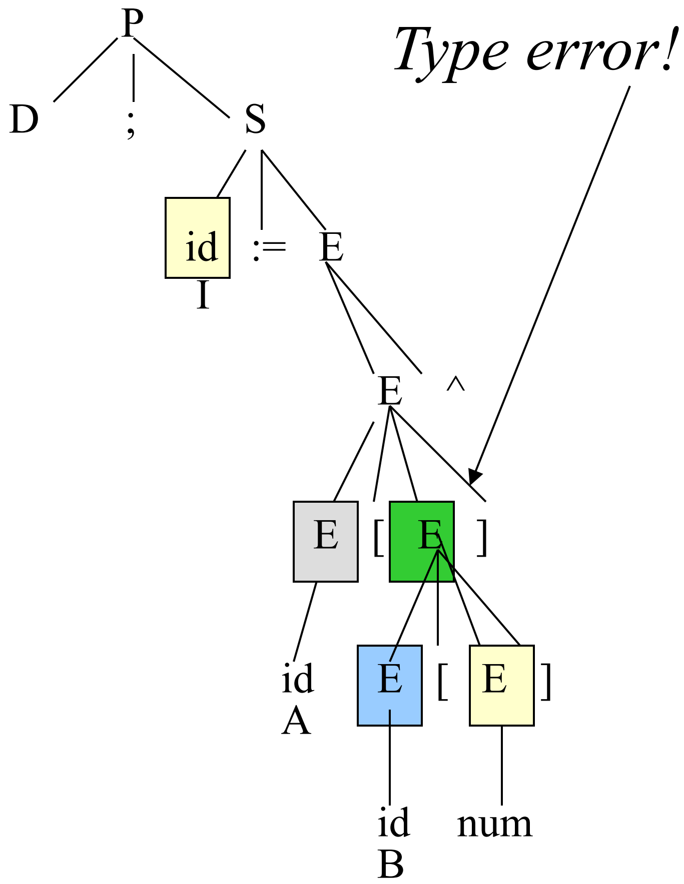
I: integer;

A: array[20] of integer;

B: array[20] of ^integer;

I := A[B[2]]^





Typechecking Statements

```
S → S1 ; S2          {if S1.type = void &  S2.type = void)
                      then S.type = void; else error(); }
S → id := E          {if lookup(id.name) = E.type
                      then S.type  = void; else error(); }
S → if E then S1     {if E.type = boolean and S1.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 → S1 ; S2                {S.type = S2.type;}
S → id := E                 {if lookup(id.name) = E.type
                              then S.type  = E.type;
                              else error();
                             }
S → if E then S1 else S2   {if (E.type = boolean & S1.type = S2.type)
                              then S.type = S1.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

1. Base Types
2. Compound/Constructed Types
3. Type Equivalence
4. Inference Rules (Typechecking)
5. …

Different languages make different choices!

1. Base (built-in) types

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

2. Constructed Types

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

3. Type Equivalence

Problem: When in E1 .type == E2 .type?

- We need a precise definition for type equivalence
- Interaction between type equivalence and type representation
- Example:
  - type vector = array [1..10] of real
  - type weight = array [1..10] of real
  - var x, y: vector; z: weight
- **Name Equivalence**: When they have the same name.
  - x, y have the same type; z has a different type.
- **Structural Equivalence**: When they have the same structure.
  - x, y, z have the same type.

Structural Equivalence

- $S \equiv T$ if:
- S and T are the same basic type;
- $S = array(S1)$, $T = array(T1)$, and $S1 \equiv T1$.
- $S = pointer(S1)$, $T = pointer(T1)$, and $S1 \equiv T1$.
- $S = tuple(S1, S2)$, $T = tuple(T1,T2)$, and $S1 \equiv T1$ and $S2 \equiv T2$.
- $S = arrow(S1, S2)$, $T = arrow(T1,T2)$, and $S1 \equiv T1$ and $S2 \equiv T2$.

Implementing Structural Equivalence

To determine whether two types are structurally equivalent, we 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) -- casting
  - Implicit → built-in (C int/char) -- coercions
- **Overloading** – meaning based on context
  - Built-in – addition operator
  - Extracting meaning – parameters/context
- Objects (inheritance)
- Polymorphism (e.g. in Java a parent class reference is used to refer to a child class object)

Part II: Scope

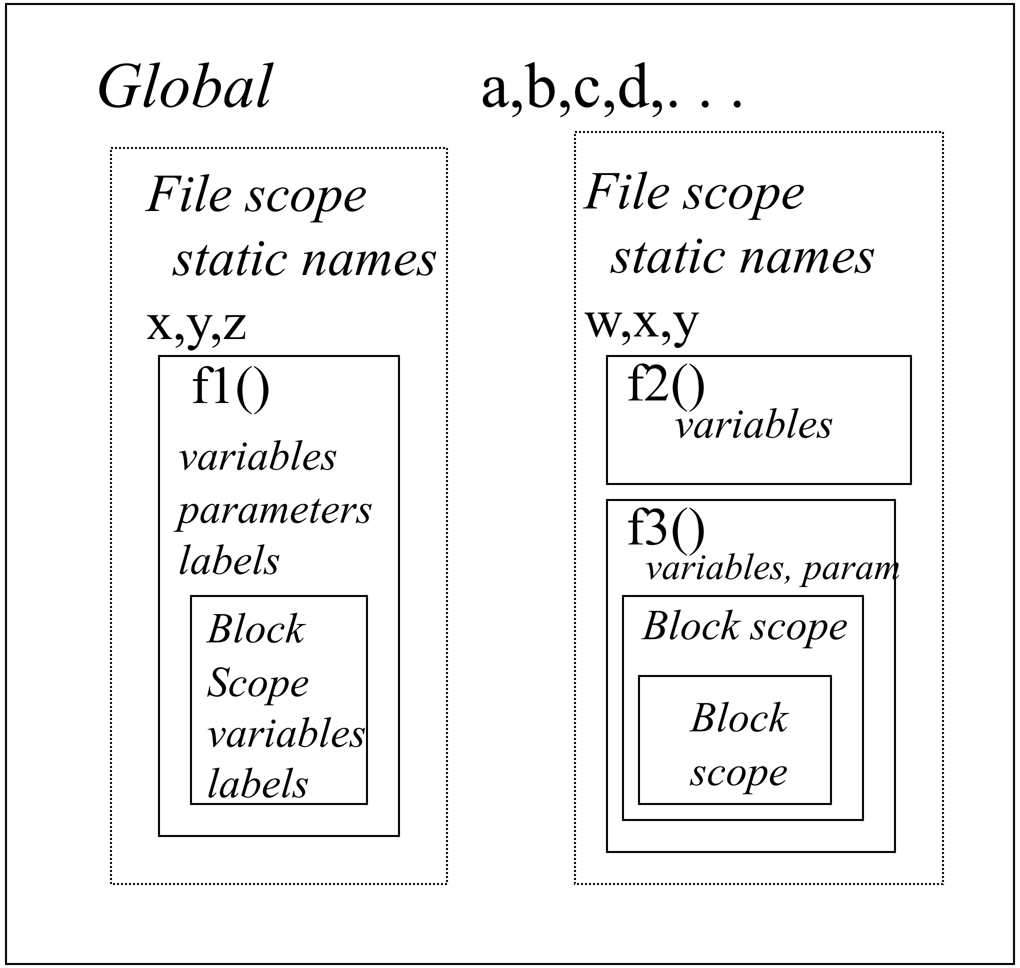
Scope

- In most languages, a complete program will contain several different **namespaces** or **scopes**.
- Different languages have different rules for namespace definition
- 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.

```
int OMG() {
        int x= 137;
        {
                string x= "Scope!"
                if (float x= 0)
                        double x= x;
        }
        if (x== 137) cout << "Y";
}
```

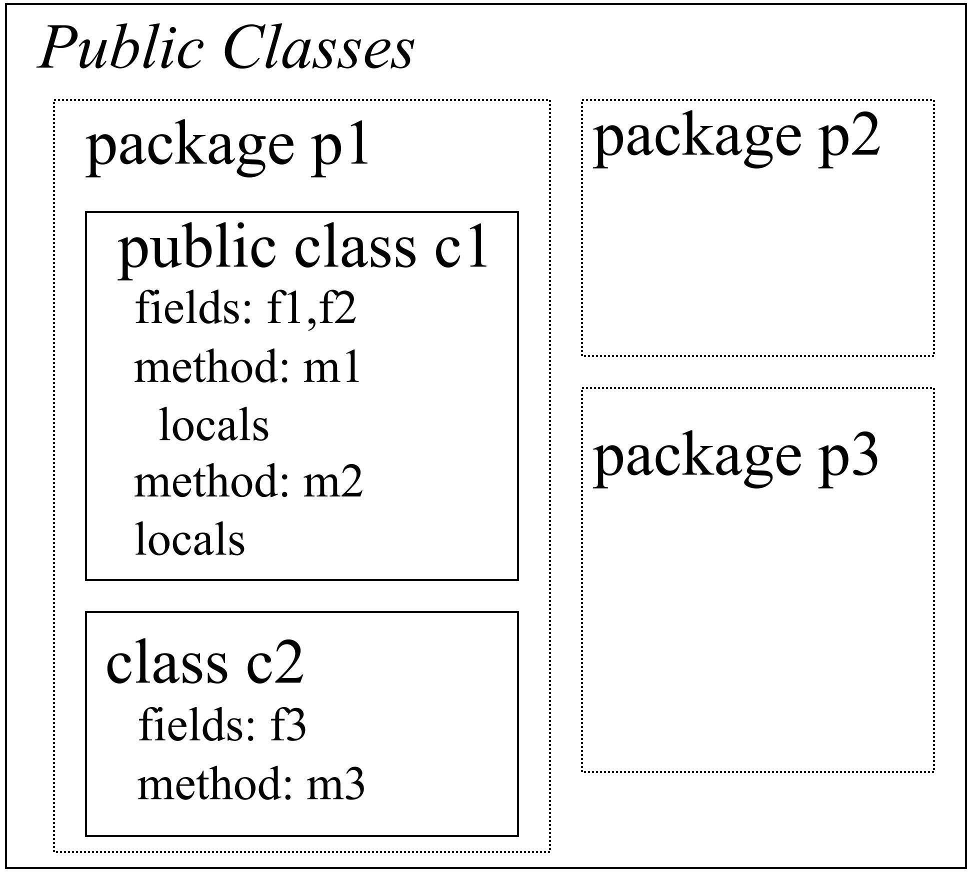
C Name Space

- 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.



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

...

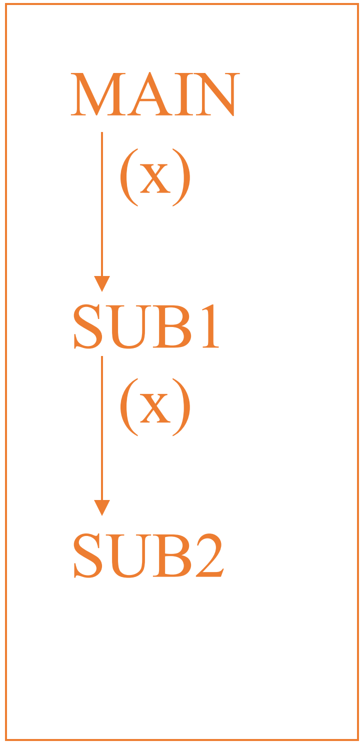
MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

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

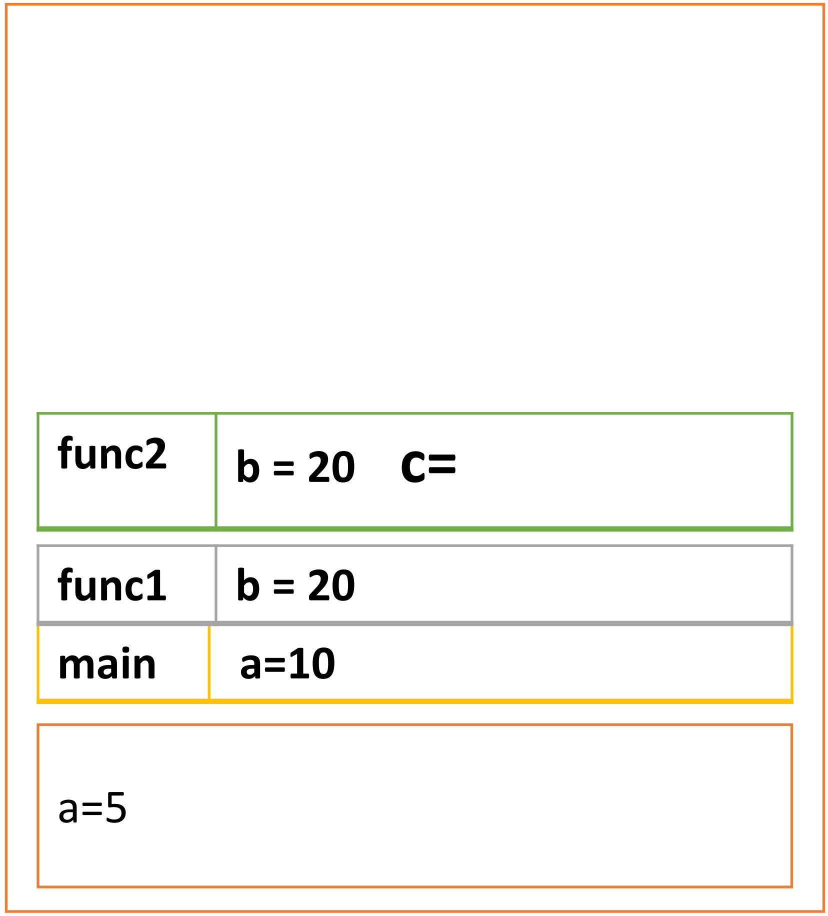
For dynamic scoping,
it is sub1's x

Dynamic Scope Example

```
int a=5;
int main ()
{
int a=10;
a=func1(a);
printf("%d",a);
}
int func1(int b)
{
b=b+10;
b=func2(b);
return b;
}
```

```
int func2(int b)
{
int c=a+b;
return c;
}
```

runtime stack

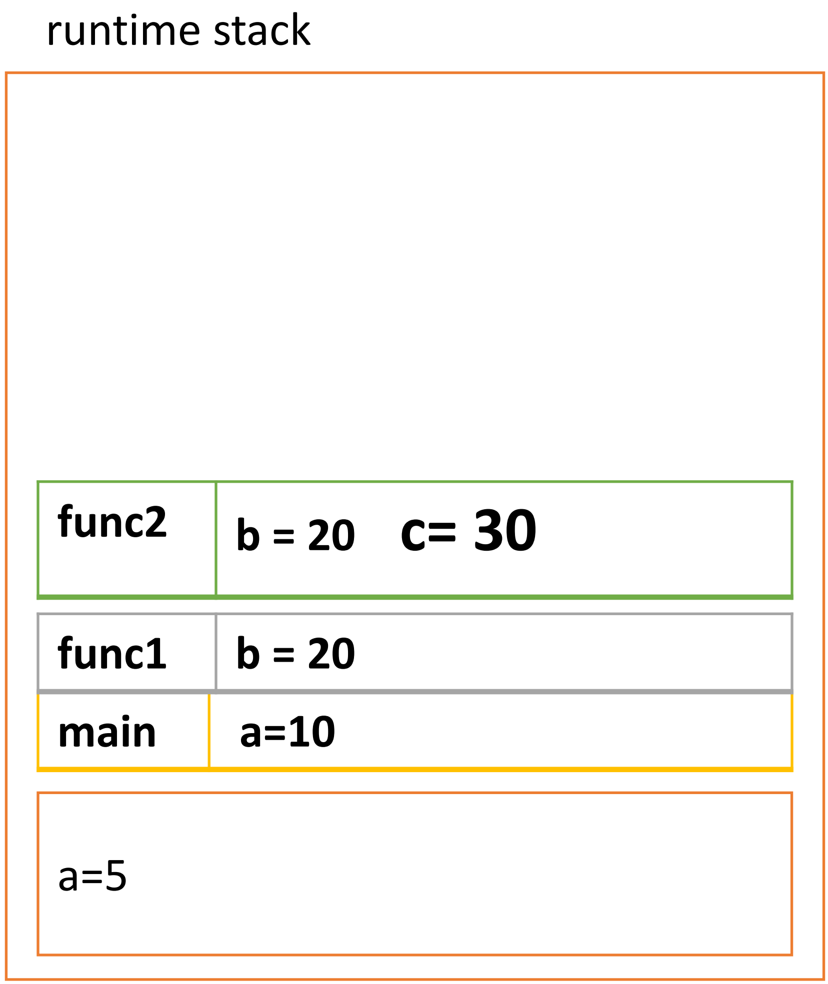


Output:
30

Dynamic Scope Example

```
int a=5;
int main ()
{
int a=10;
a=func1(a);
printf("%d",a);
}
int func1(int b)
{
b=b+10;
b=func2(b);
return b;
}
```

```
int func2(int b)
{
int c=a+b;
return c;
}
```



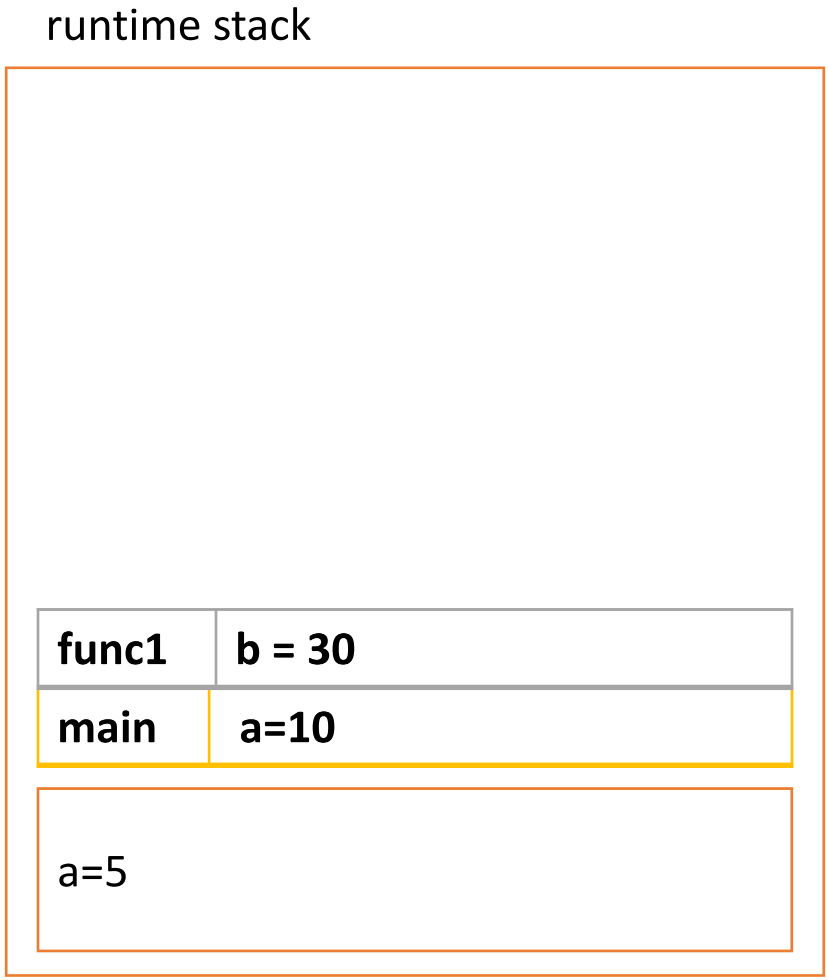
Output:
30

Dynamic Scope Example

```
int a=5;
int main ()
{
int a=10;
a=func1(a);
printf(“%d”,a);
}
int func1(int b)
{
b=b+10;
b=func2(b);
return b;
}
```

```
int func2(int b)
{
int c=a+b;
return c;
}
```

Output:
30

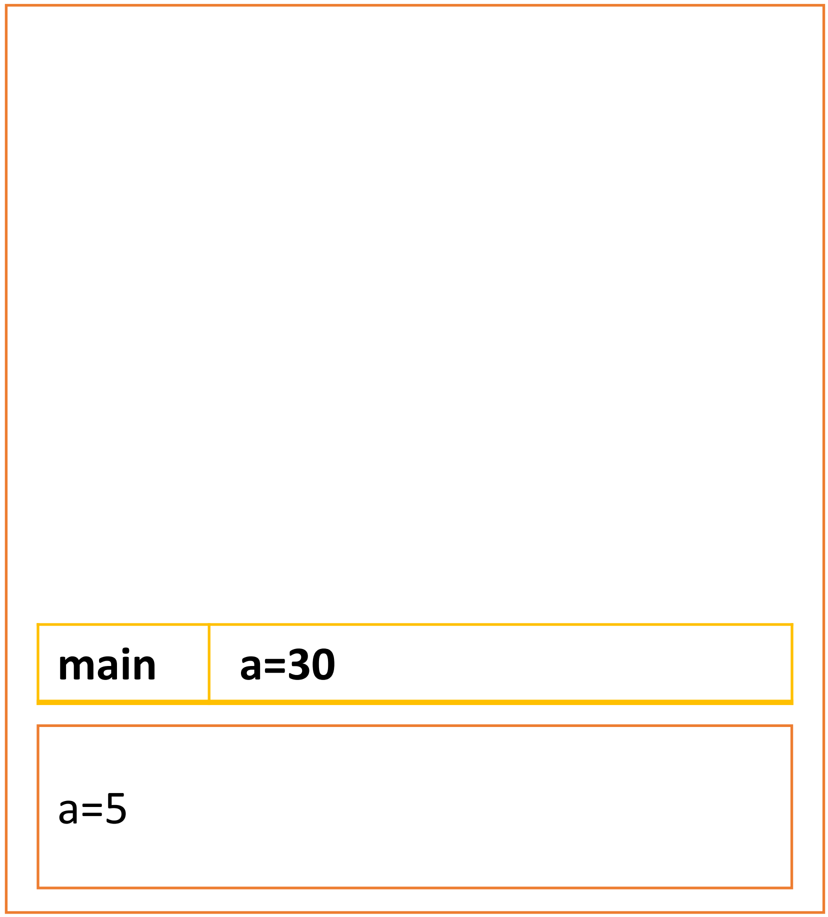


Dynamic Scope Example

```
int a=5;
int main ()
{
int a=10;
a=func1(a);
printf(“%d”,a);
}
int func1(int b)
{
b=b+10;
b=func2(b);
return b;
}
```

```
int func2(int b)
{
int c=a+b;
return c;
}
```

runtime stack



Output:
30

```c
int a,b;

void print() {
    printf("%d %d", a, b);
}

int fun1() {
    int a, c;
    a = 0; b = 1; c = 2;
    return c;
}

void fun2() {
    int b;
    a = 3; b = 4;
    print();
}

int main() {
    a = fun1();
    fun2();
}
```

What is the output

1. Static scoping

2. Dynamic scope

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:

- **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

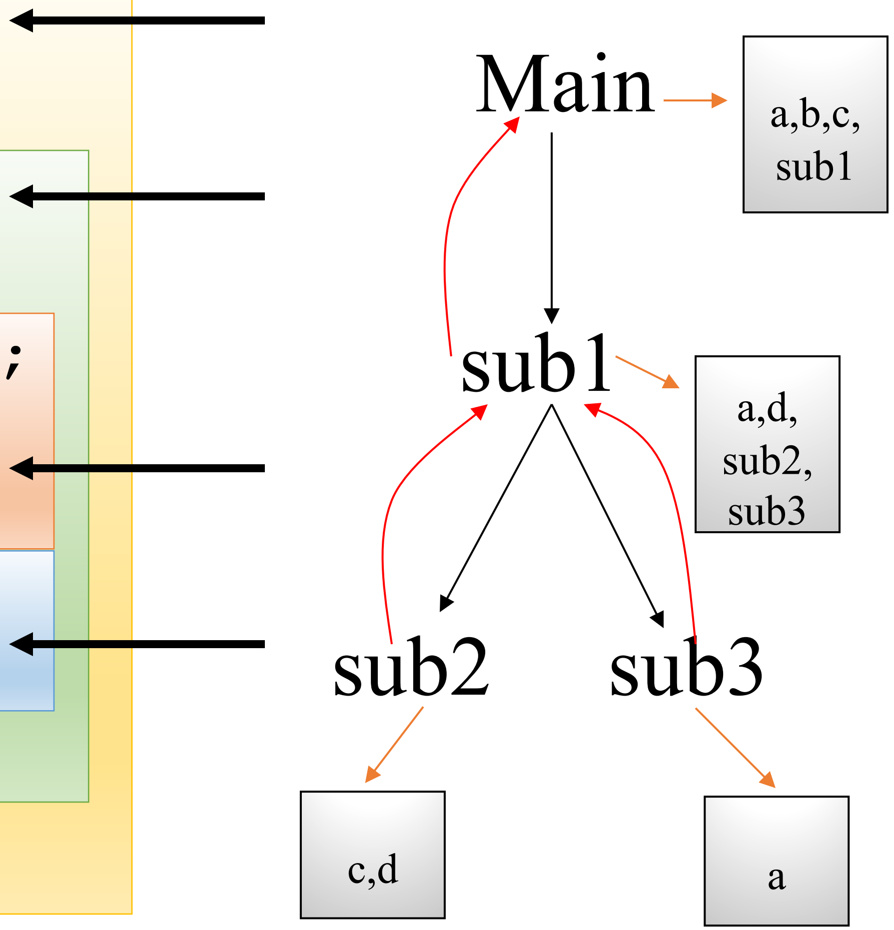
A tree?

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

```
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
```