Type Checking

Type Checking

- A compiler has to do semantic checks in addition to syntactic checks.
- Semantic Checks
  - Static – done during compilation
  - Dynamic – done during run-time
- *Type checking* is one of these static checking operations.
  - we may not do all type checking at compile-time.
  - Some systems also use dynamic type checking too.

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.

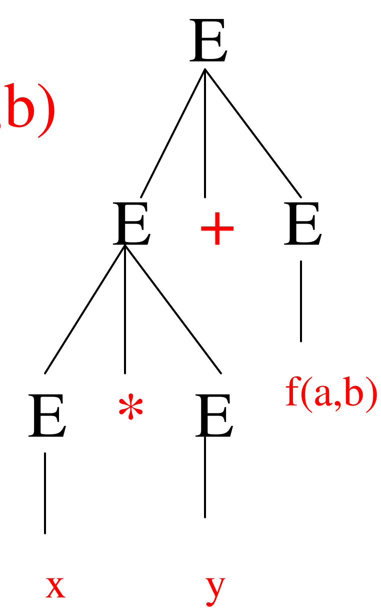
Type Systems

- A *sound* type system eliminates run-time type checking for type errors.
- A programming language is *strongly-typed*, if every program its compiler accepts will execute without type errors.
  - In practice, some of type checking operations are done at run-time (so, most of the programming languages are not strongly-typed).
  - Ex: int x[100]; … x[i] → most of the compilers cannot guarantee that i will be between 0 and 99

Type checking

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? Would the operation be "integerMUL" or "floatingMUL"
- 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

Components of a Type System

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

Different languages make different choices!

Types

- Each language has its own notions of “type”
- **Basic Types** (also called “primitive types”)
  - **integer, real, character, boolean**
- **Constructed Types**
  Built from other types...

array of ...	**int [100] a**
record { ... }	
pointer to ...	**int *p**
function (...) → …	**int (* foo) (...) {...}**

- We must represent types within the compiler.
- Might want a little language of “***type expressions***”.
  - To make explicit the universe of all possible types.

Basic Types

Each has a name

integer
real
boolean
char
...
void
type_error

Each basic type is a set of values.
Each type will have several Predefined operators on the values

Void

A type with zero values
Used for typing functions

Type_Error

Used to deal with semantic (type) errors (not really a type)

Constructed types

A *type expression* is either a basic type or is formed by applying an operator called ***type constructor*** to other type expressions.

A type name: a name can be used to denote a type expression.

Arrays: If T is a type expression, then **array(I,T)** is a type expression where I denotes index range.

For example the declaration:

var A: array[0..99] of integers

associates type expr. **array(0..99,integer)** to A

Pointer: T is a type expression, then **pointer(T)** is a type expression.

For example: var p: ↑ integer; OR var p: ^integer; OR var p: int *p;

associates the type expression **pointer(integer)** to p

Product Type (tuple types)

products: If T1 and T2 are type expressions, then their Cartesian product $T1 \times T2$ is a type expression.

Each tuple object consists of several component values.

- Each component value has a different type. (Similar to record types).
- Component values are identified by position, not name.

Notation #1:

var t1: integer × boolean;

t2: real × real × real × real;

Notation #2:

var t1: (integer, boolean);

t2: (real, real, real, real);

To specify a tuple:

- **t1 = <6,true>; or, t1 = (6,true); or, t1 = [6,true];**

To access the component values:

- **x = t2.3;   x = third(t2);**

Record ("struct") type

The record type constructor will be applied to a tuple formed from field names and field types.

Type row = record

address : integer;

lexeme: array[1..15] of char

end;

Var table: array[1..101] of row;

Declares the type name row representing the type expression

record((address $\times$ integer) $\times$ (lexeme $\times$ array(1..15,char))

Function type

- We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression $D \rightarrow R$ where D are R type expressions.

- Usually we use the notation,

function (*DomainTypes***) returns** *RangeType*

g: function (char, char, char, char) returns ↑ integer;

Type expr: **char × char × char × char → pointer(integer)**

Syntax Directed Type Checking

Consider the following simple language

$P \rightarrow D \; ; \; S$

$D \rightarrow D \; ; \; D \;|\; id : T$

$T \rightarrow integer \;|\; array \; [\; num \;] \; of \; T \;|\; \hat{}T \;|\; T \rightarrow T \;|\; T \times T$

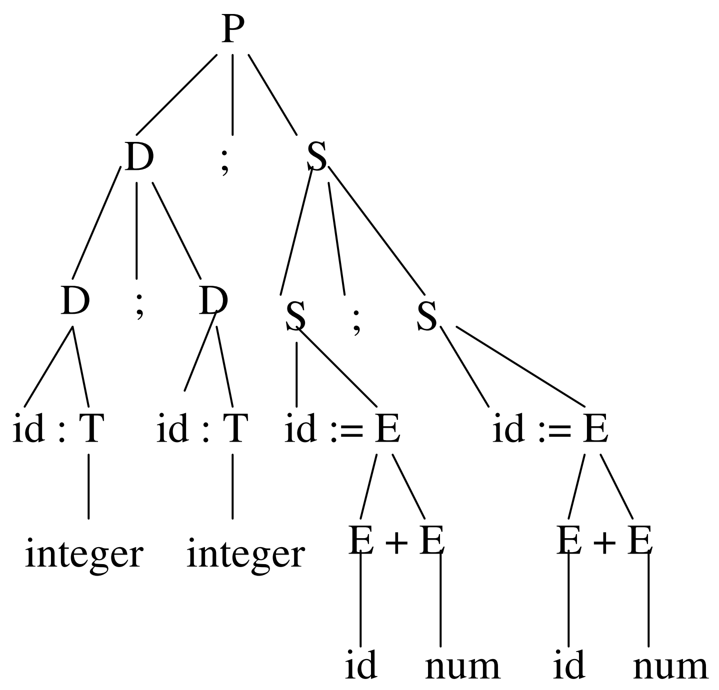
$S \rightarrow S \; ; \; S \;|\; id := E \;|\; if \; E \; then \; S \;|\; while \; E \; do \; S$

$E \rightarrow num \;|\; id \;|\; E + E \;|\; E \; [\; E \;] \;|\; E\hat{} \;|\; E \; (\; E \;)$

How can we type-check strings in this language?

Example of language

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



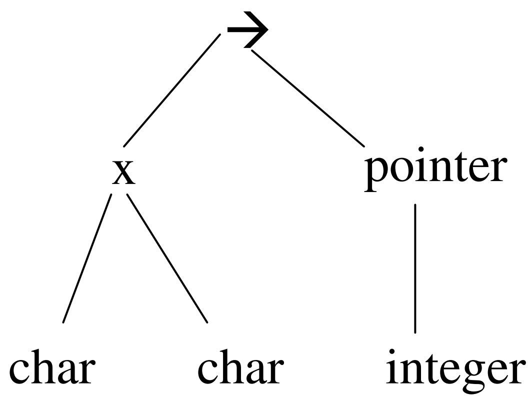
Processing Declarations

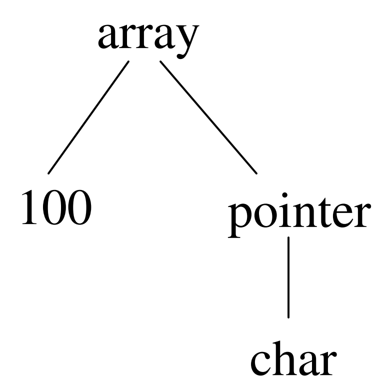
Put info into
the symbol table

$D \rightarrow D ; D$	
$D \rightarrow id : T$	{insert(id.name,T.type);}
$T \rightarrow integer$	{T.type = integer;}
$T \rightarrow array [num] of T_1$	{T.type = array(1..num.val, T_1.type); }
$T \rightarrow \hat{} T_1$	{T.type = pointer(T_1.type);}
$T \rightarrow T_1 \times T_2$	{T.type = product(T_1.type, T_2.type);}
$T \rightarrow T_1 \rightarrow T_2$	{T.type = function(T_1.type, T_2.type);}

Accumulate information about
the declared type

Can use Trees (or DAGs) to Represent Types





char x char ➔ pointer(integer)

array[100] of pointer(char)

Build data structures while we parse

Example

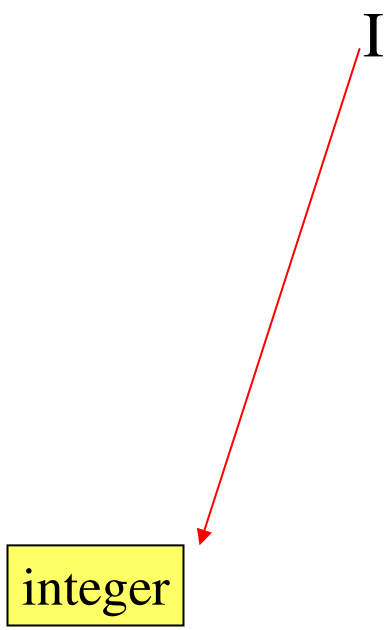
I: integer;

A: array[20] of integer;

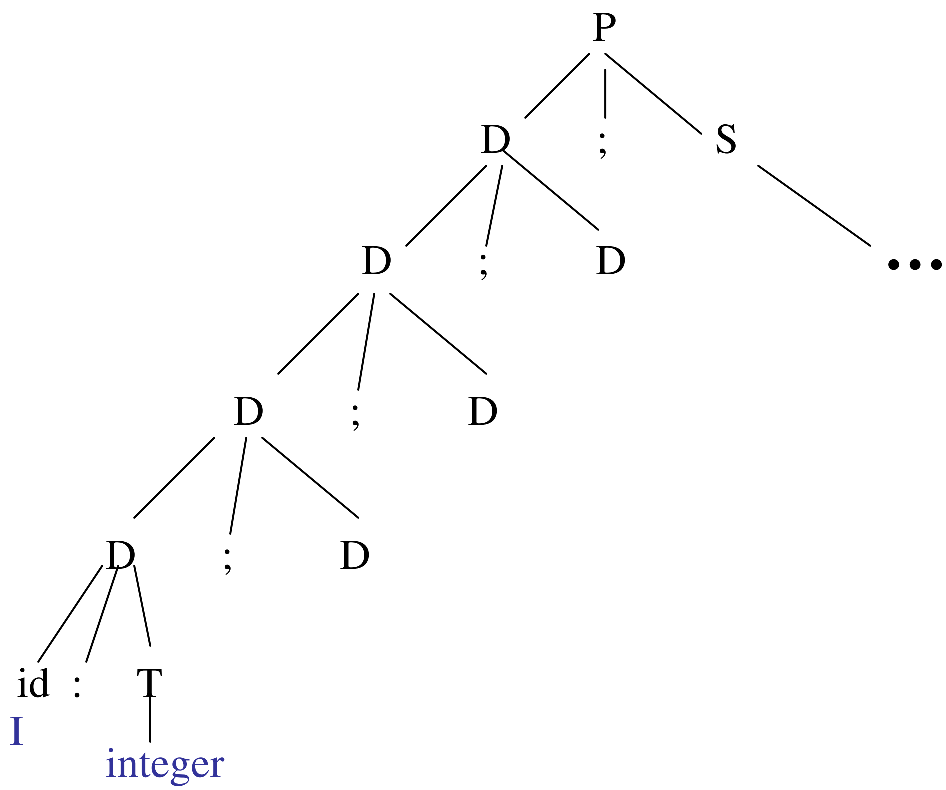
B: array[20] of ^integer;

F: ^integer → integer;

I := F(B[A[2]])



Parse Tree



Example

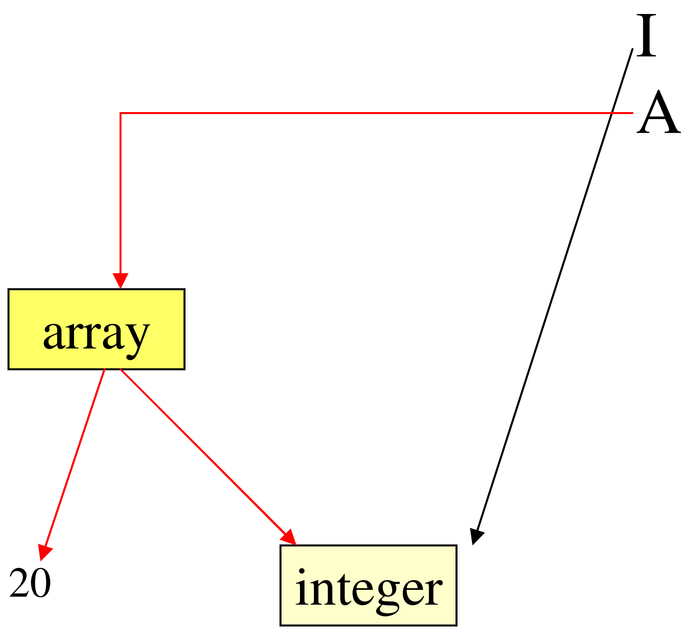
I: integer;

A: array[20] of integer;

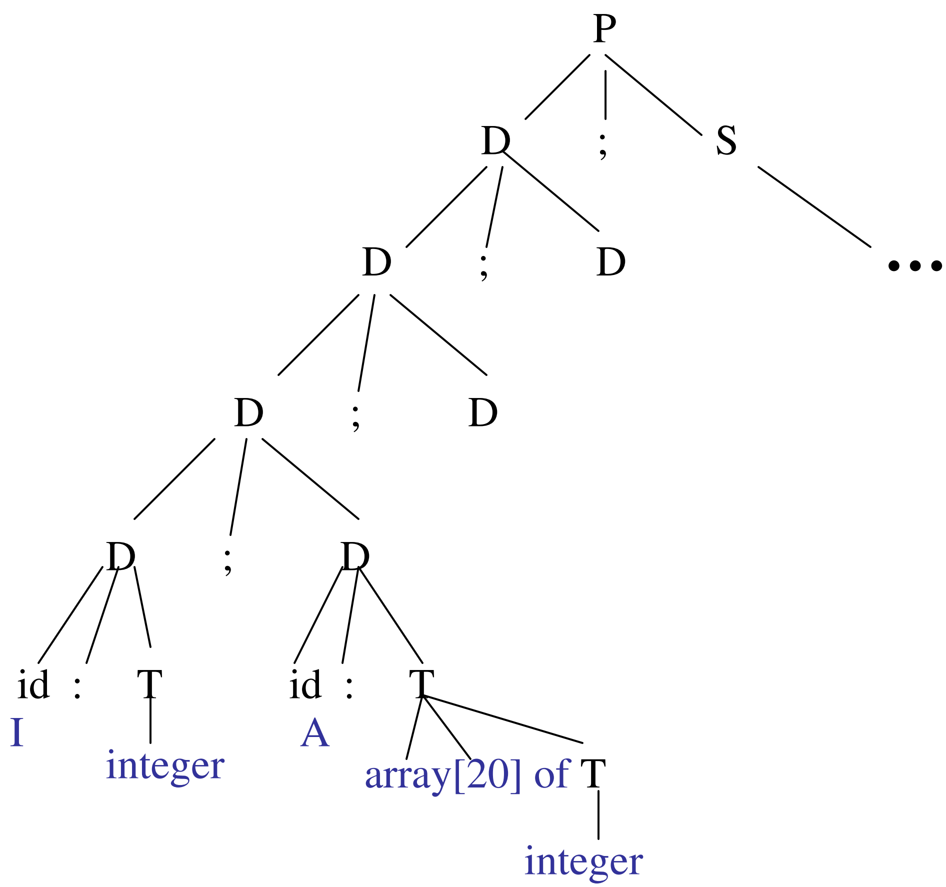
B: array[20] of ^integer;

F: ^integer → integer;

I := F(B[A[2]])



Parse Tree



Example

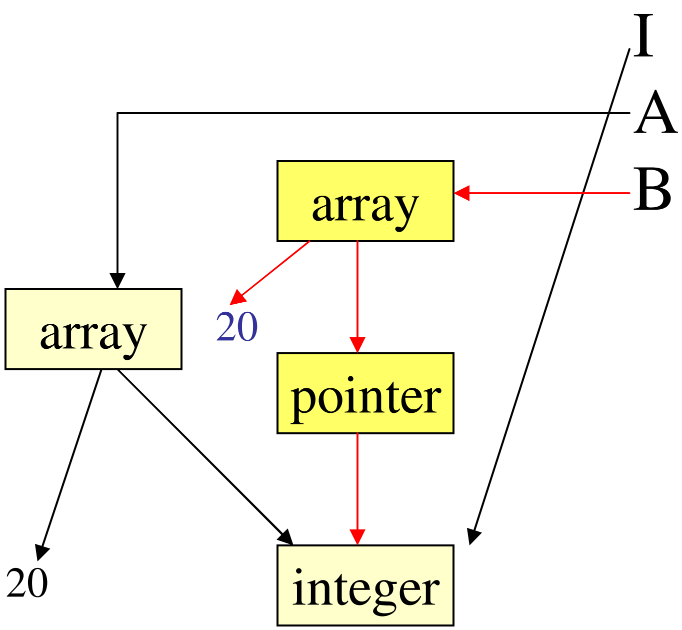
I: integer;

A: array[20] of integer;

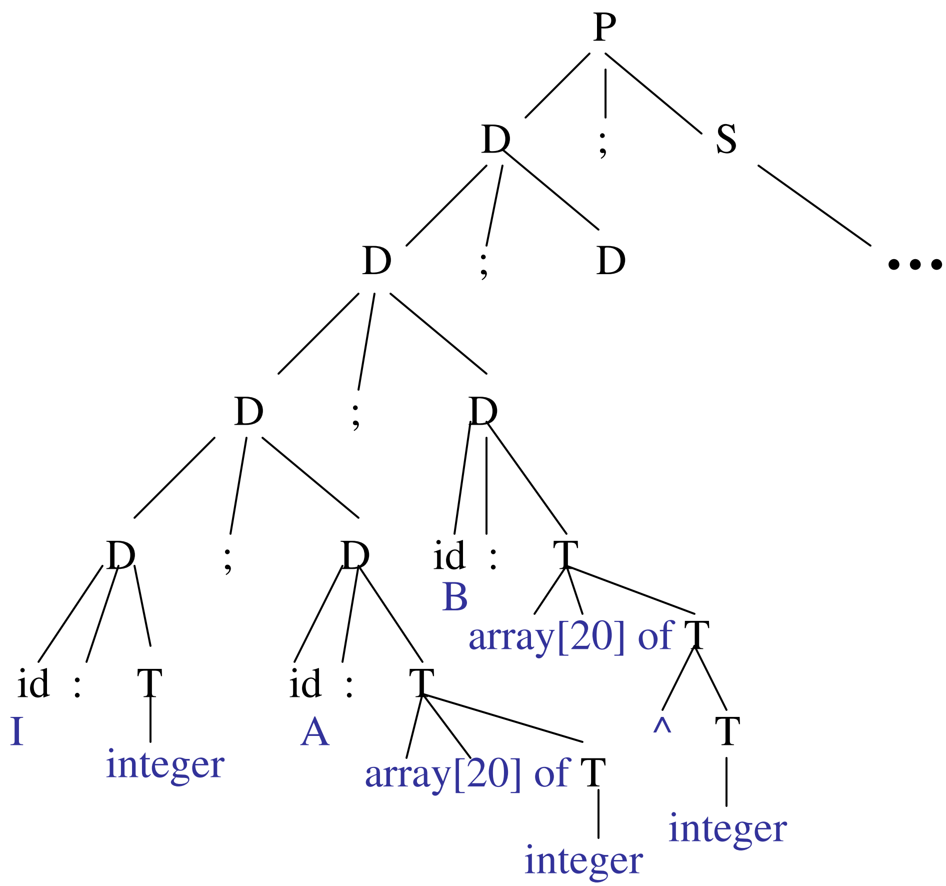
B: array[20] of ^integer;

F: ^integer → integer;

I := F(B[A[2]])



Parse Tree



Example

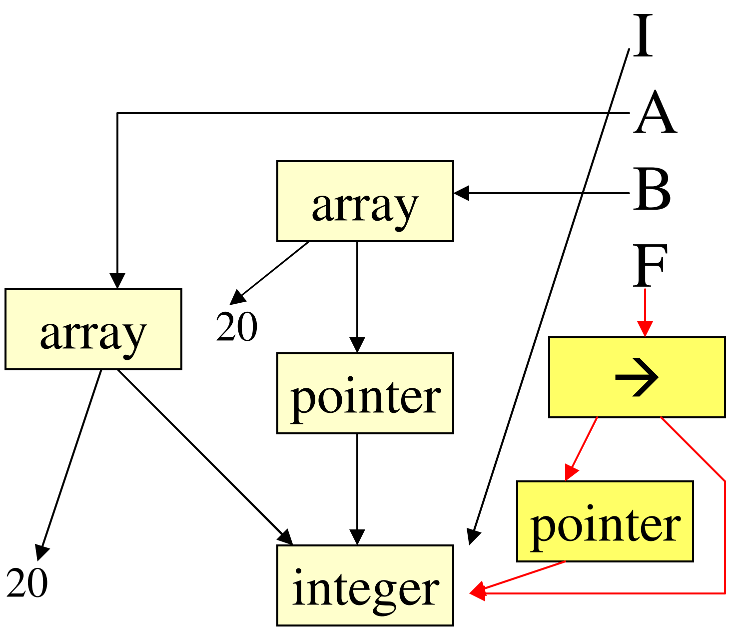
I: integer;

A: array[20] of integer;

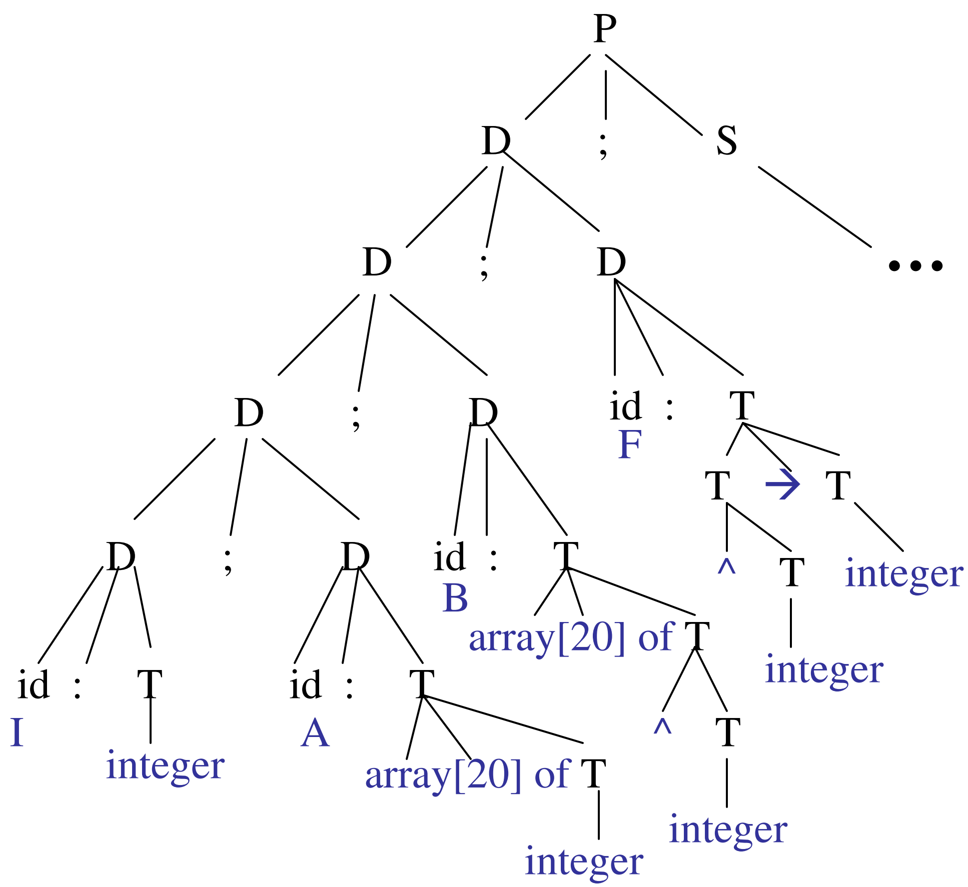
B: array[20] of ^integer;

F: ^integer → integer;

I := F(B[A[2]])



Parse Tree



Type Checking of Expressions

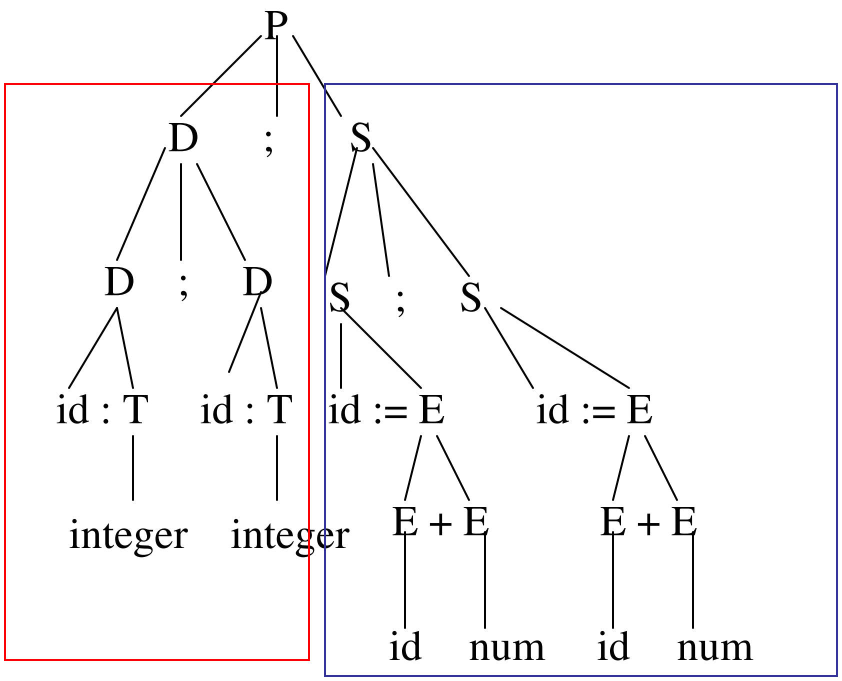
$E \rightarrow$ **id**	{ E.type=lookup(id.entry) }
$E \rightarrow$ **charliteral**	{ E.type=char }
$E \rightarrow$ **intliteral**	{ E.type=int }
$E \rightarrow$ **realliteral**	{ E.type=real }
$E \rightarrow E_1 + E_2$	{ if (E_1.type=int and E_2.type=int) then E.type=int else if (E_1.type=int and E_2.type=real) then E.type=real else if (E_1.type=real and E_2.type=int) then E.type=real else if (E_1.type=real and E_2.type=real) then E.type=real else E.type=type-error }
$E \rightarrow E_1 [E_2]$	{ if (E_2.type=int and E_1.type=array(s,t)) then E.type=t else E.type=type-error }
$E \rightarrow E_1 \uparrow$	{ if (E_1.type=pointer(t)) then E.type=t else E.type=type-error }
$E \rightarrow E_1 (E_2)$	{ if (E_1.type = $T_1 \rightarrow T_2$ & E_2.type = T_1) then E.type = T_2; else …}

Example

i: integer; j: integer;

i := i + 1;

j := i + 1



Example

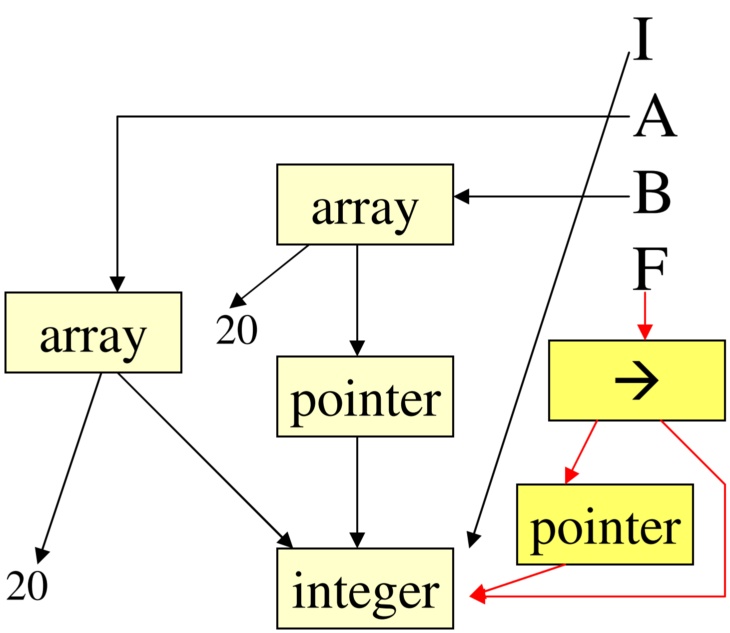
I: integer;

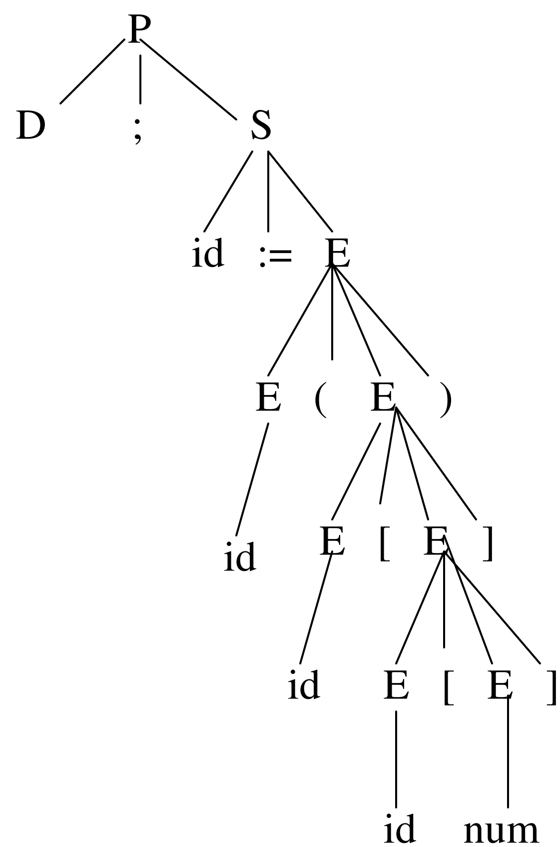
A: array[20] of integer;

B: array[20] of ^integer;

F: ^integer → integer;

I := F(B[A[2]])





Example

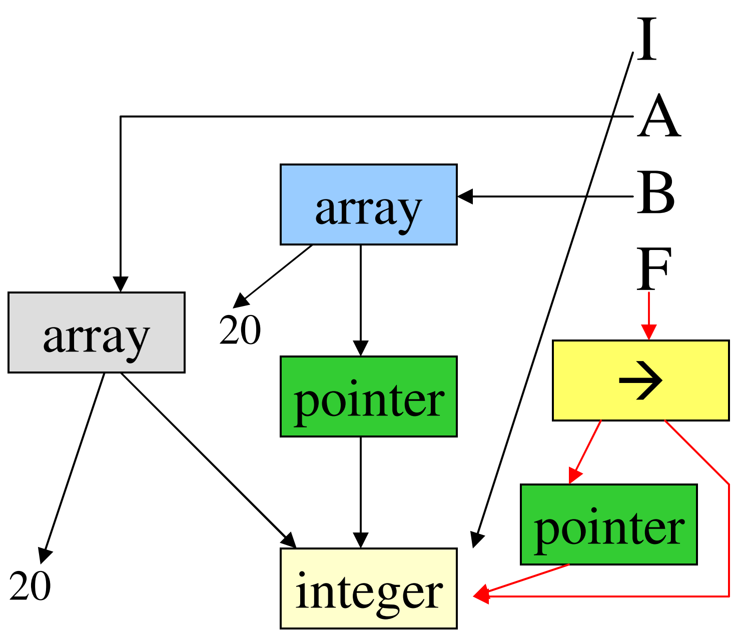
I: integer;

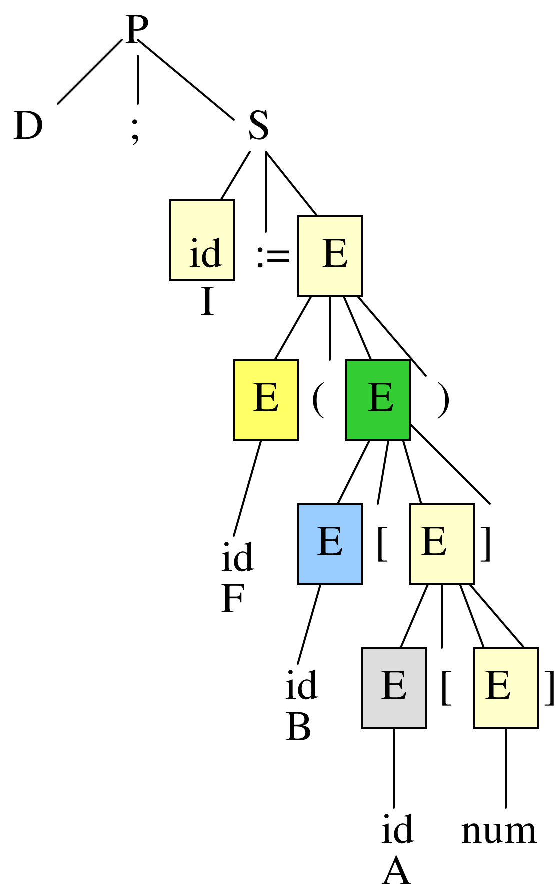
A: array[20] of integer;

B: array[20] of ^integer;

F: ^integer → integer;

I := F(B[A[2]])





Example

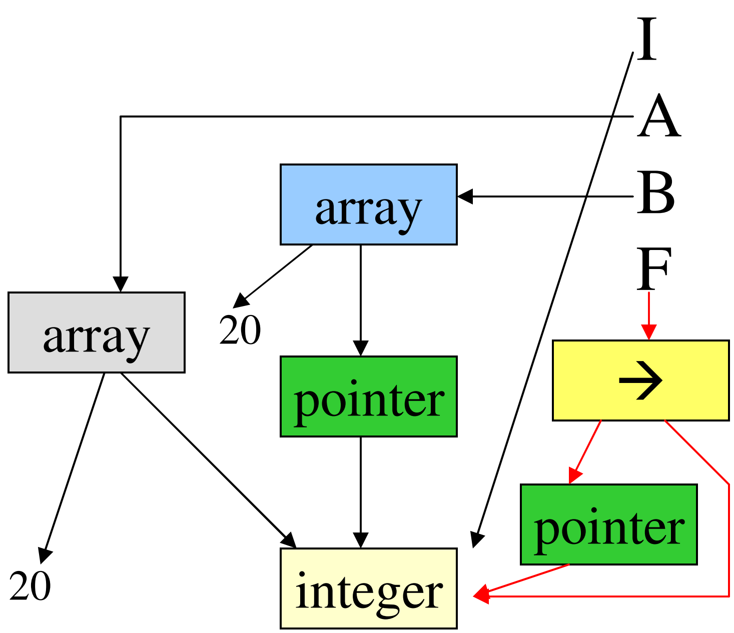
I: integer;

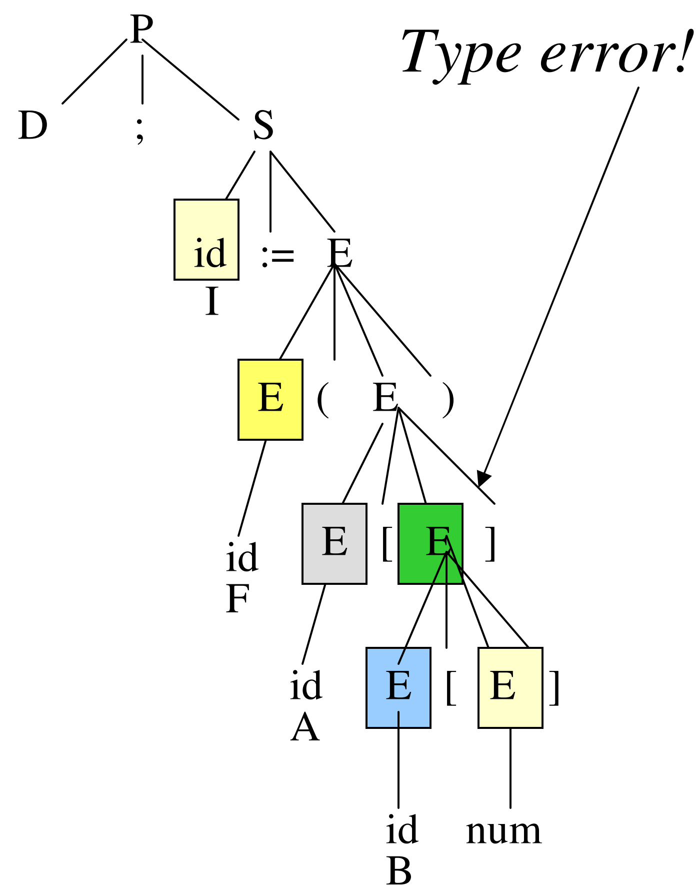
A: array[20] of integer;

B: array[20] of ^integer;

F: ^integer → integer;

I := F(A[B[2]])





Type Checking of Statements

$S \rightarrow$ **id** $= E$ { if (id.type=E.type then S.type=void
else S.type=type-error }

$S \rightarrow$ if E then S_1 { if (E.type=boolean then
S.type=S_1.type
else S.type=type-error }

$S \rightarrow$ while E do S_1 { if (E.type=boolean then S.type=S_1.type
else S.type=type-error }

In this case, we assume that statements do not have values so we assign void types

Type Checking of Functions

$$E \rightarrow E_1 \; (\; E_2 \;) \; \{ \; \text{if } (E_2.\text{type}=s \text{ and } E_1.\text{type}=s\rightarrow t) \text{ then}$$

$$E.\text{type}=t$$

$$\text{else } E.\text{type}=\text{type-error} \; \}$$

Ex: int f(double x, char y) { ... }

f: $\text{double} \times \text{char} \rightarrow \text{int}$

argument types   return type

Approach To Static Type Checking

- Need to describe types
  - A representation of types
- Associate a type with each variable.
  - The variable declaration associates a type with a variable.
  - This info is recorded (in the symbol table).
- Associate a type with each expression
  - ...and each sub-expression.
- Work bottom-up
  - The type is a "synthesized" attribute
- Check operators
  - **expr1 + expr2**
    - Is the type of the expressions "integer" or "real"?
- Check other places that expressions are used
  - **LValue := Expr ;**
    - Is the type of "expr" equal to the type of the L-Value?
  - **if (expr) ...**
    - Is the type of the expression "boolean"?

Untyped languages

Either:

Single type that contains all values

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

OR:

- There may be different types of data (integer, float, pointers, etc.)
- The programmer says which operations to use (iadd, fadd, ...)
- A type is not associated with each variable.
- If the programmer makes mistakes, the results are wrong.

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.

Type Equivalence

- How do we know that two type expressions are equal?

- Two types of equivalence: Structural and Name

  Type A = Bool

  Type B = Bool

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

Type Equivalence

What does it mean to say "type of operand 1" = "type of operand 2"?

```
type T1 is record
               f: int;
               g: real;
             end;
     T2 is record
               f: int;
               g: real;
             end;
     T3 is T2;
var x: T1,
    y: T2,
    z: T3;
...
x := y;
```

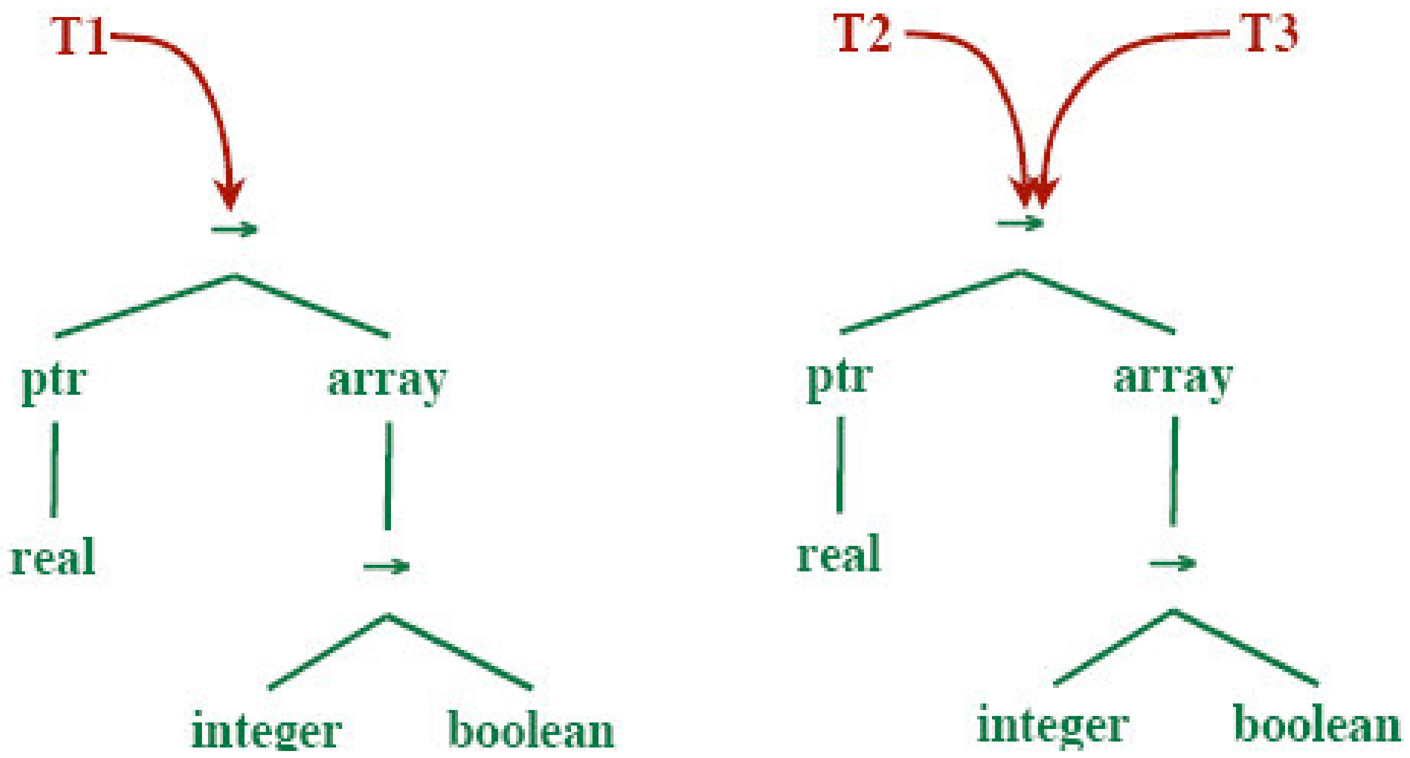
Is the type of "x" the same as the type of "y"?
Is the type of "y" the same as the type of "z"?

Type Equivalence

Types are represented as trees.

Types may be named.

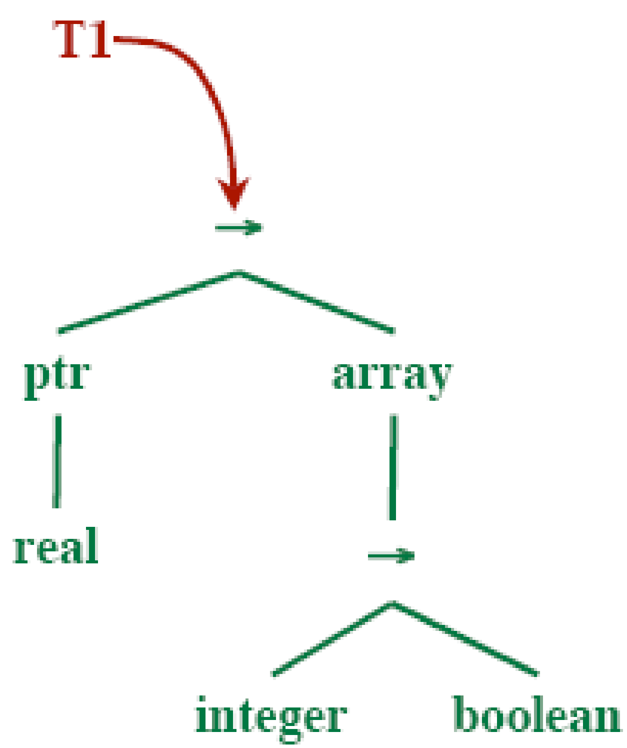
type T1 is ... ;



Type Equivalence

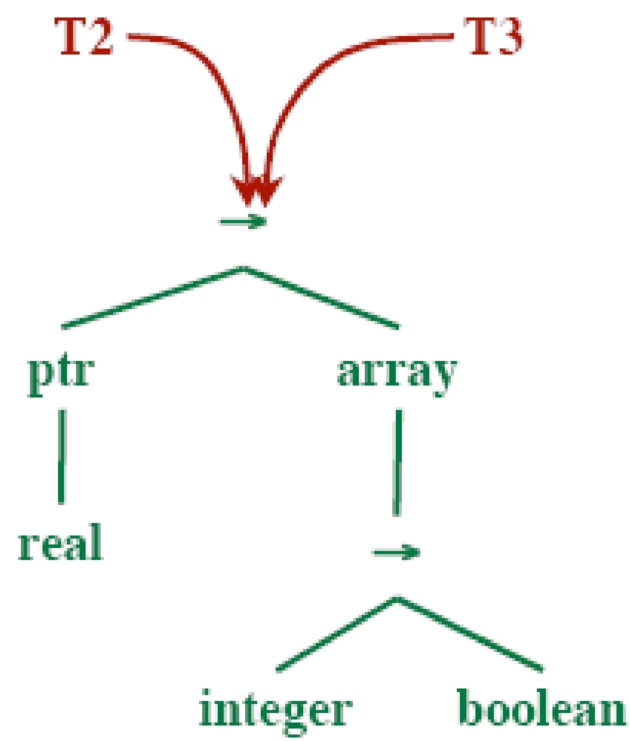
Structural Equivalence

Are the trees equivalent?
Isomorphic (same shape, same nodes)
Must walk the trees to check



Name Equivalence

Are they the same tree?
Compare pointers



Testing Structural Equivalence

```
function typeEquiv (s, t) returns boolean
  if (s and t are the same "basic" type) then
    return true
  elseif (s = "array of s1") and (t = "array of t1") then
    return typeEquiv (s1,t1)
  elseif (s = "s1 × s2") and (t = "t1 × t2") then
    return typeEquiv (s1,t1) and typeEquiv (s2,t2)
  elseif (s = "ptr to s1") and (t = "ptr to t1") then
    return typeEquiv (s1,t1)
  elseif (s = "s1 → s2") and (t = "t1 → t2") then
    return typeEquiv (s1,t1) and typeEquiv (s2,t2)
  else
    return false
  endIf
endFunction
```

Names for Type Expressions

- In some programming languages, we give a name to a type expression, and we use that name as a type expression afterwards.

```
type link = ↑ cell;
var p,q : link;
var r,s : ↑ cell
```

p,q,r,s have same types ?

- How do we treat type names?
  - Get equivalent type expression for a type name (then use structural equivalence), or
  - Treat a type name as a basic type.

Cycles in Type Expressions

```
type link = ↑ cell;
type cell = record
              x : int,
              next : link
            end;
```

- We cannot use structural equivalence if there are cycles in type expressions.
- We have to treat type names as basic types.
  - ➔ but this means that the type expression `link` is different than the type expression ↑`cell`.

Type Conversion

```
 var i: int,
        x: real;
... (x + i ) ...
```

Must convert the integer value to a real value first

Real addition (fadd) will be used

The result will be a real

Implicit Type Conversions (also called "Coercions")

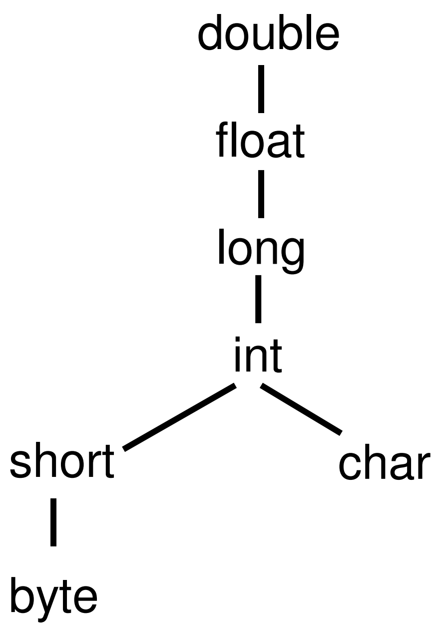
- The language definition tells when they are needed.
- Compiler must insert special code to perform the operation.

Explicit Type Conversions (also called "Casting")

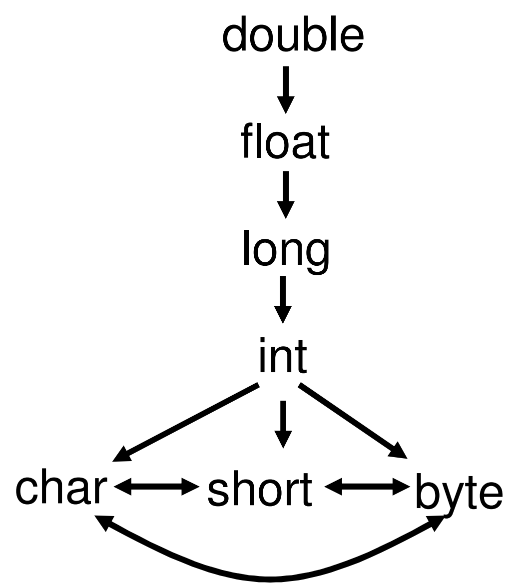
- **... (i + (int) x) ...**
- The programmer requests a specific conversion.
- The language definition tells when they allowed.
- The compiler may (or may not) need to insert special code.

Type Conversion

- Widening conversions
  - Intended to preserve information
  - Any type can be widened to a higher type
- Narrowing conversions
  - May lose information
  - Conversion between to types possible if there is a path



Widening conversions



Narrowing conversions

Type conversion

- Functions for type conversions
- **max(t_1,t_2)**: takes two types t_1 and t_2 returns the maximum (or at least upper bound) of the two types in the widening hierarchy
- **widen(a,t,w)**: generates type conversions if needed to widen an address **a** of type **t** into a type **w**.

```
Addr widen(Addr a, Type t, Type w)
        if (t=w) return a;
        else if (t=integer and w=float){
                temp=new Temp();
                gen(temp'=' '(float)' a);
                return temp;
        }
        else error
}
```

Pseudo code for function widen that uses only two types integer and float

Type Conversion into expression evaluation

$E \rightarrow E_1 + E_2$ { E.type = max(E_1.type,E_2.type);

a_1 = widen(E_1.addr,E_1.type,E.type);

a_2 = widen(E_2.addr,E_2.type,E.type);

E.addr = new Temp();

gen(E.addr '=' a1 '+' a2);}