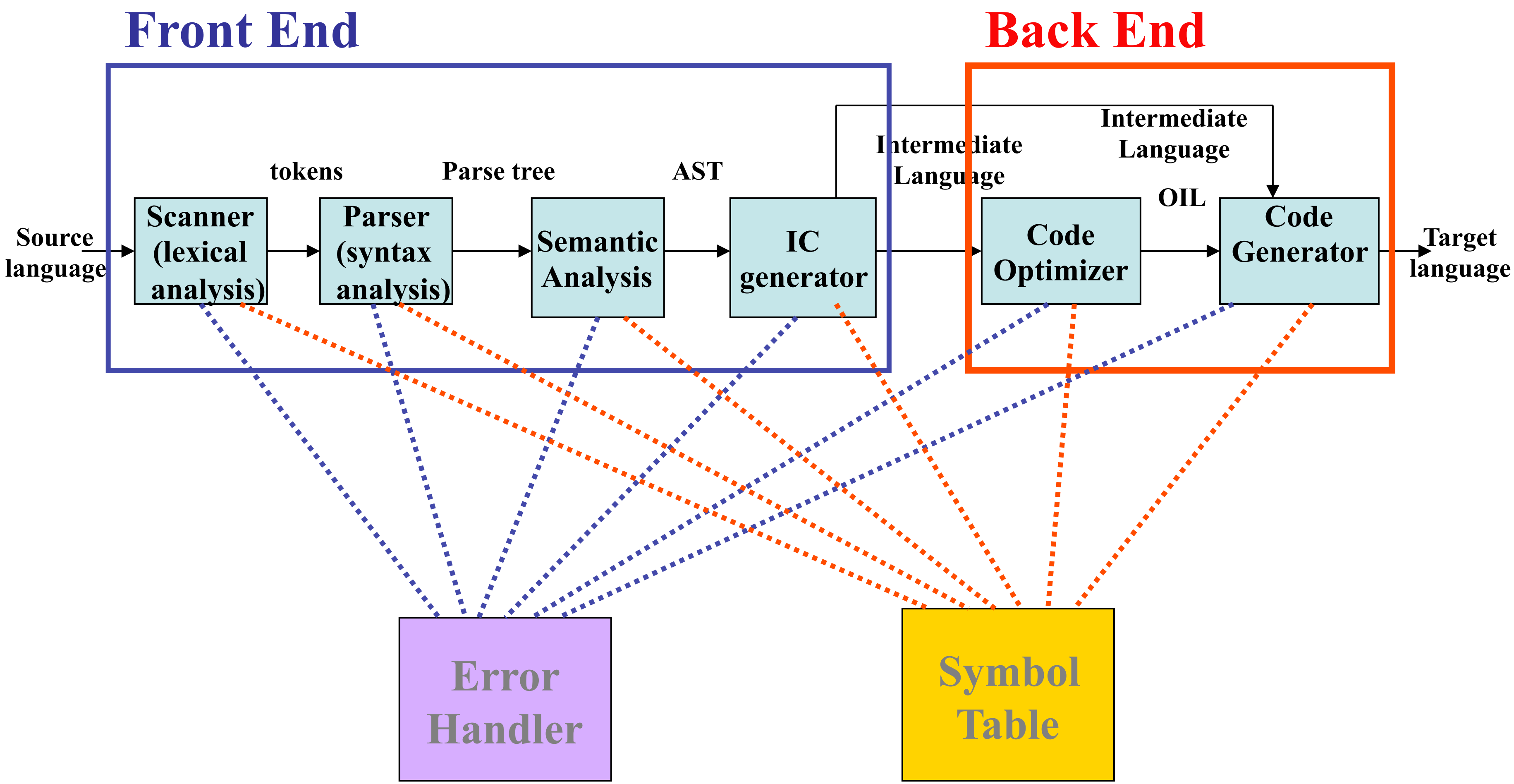
# Language Processing Systems

Prof. Mohamed Hamada

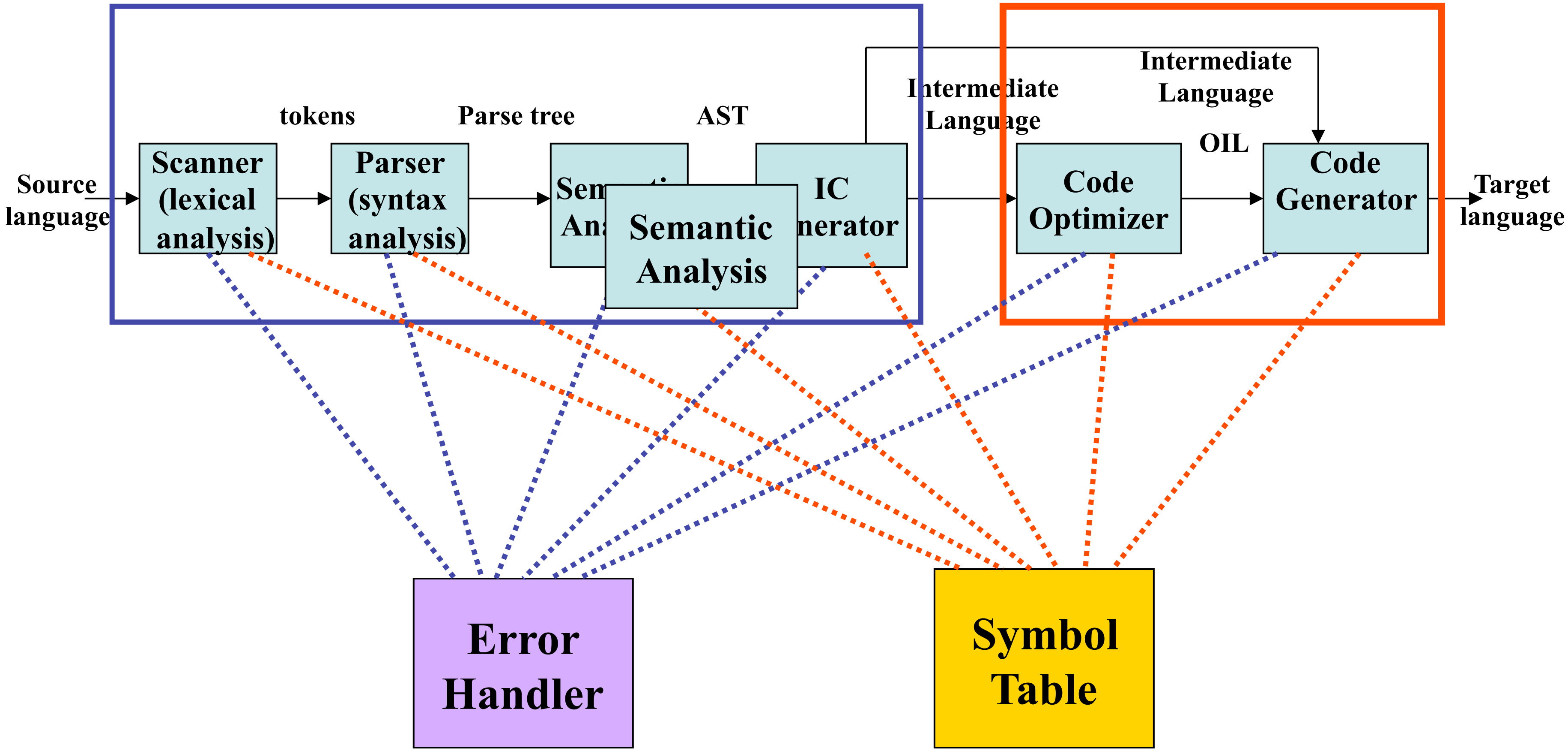
Software Engineering Lab.
The University of Aizu
Japan

# Semantic Analysis

# Compiler Architecture



# Semantic Analysis




- "Meaning"
- Type/Error Checking
- Intermediate Code Generation – abstract machine

# Semantic Analysis

- Compilers examine code to find semantic problems.
  - Easy: undeclared variables, tag matching
  - Difficult: preventing execution errors
- Essential Issues:
  - Abstract Syntax Trees (AST)
  - Scope
  - Symbol tables
  - Type checking

# Semantic Analysis

- Essential Issues:
  - Abstract Syntax Trees (AST)
  - Scope
  - Symbol tables
  - Type checking

# Semantic Analysis

- Essential Issues:
  - Abstract Syntax Trees (AST)
  - Scope
  - Symbol tables
  - Type checking

# Type Checking

# Type Checking

## Type rules:

- which types can be combined with certain operator
- assignment of expression to variable
- formal and actual parameters of a method call

A type checker is a function that maps an AST that represents an expression into its type

# Type Checking

- Checking whether the use of names is consistent with their declaration in the program

**Example**

```
int x;
x := x+1;
x.A := 1;
x[0] := 0;
```

Correct use of x

Type errors

- *Statically typed languages*: done at compile time, not at run time
- Need to remember declarations
  - Symbol Table

# Type Checking - example

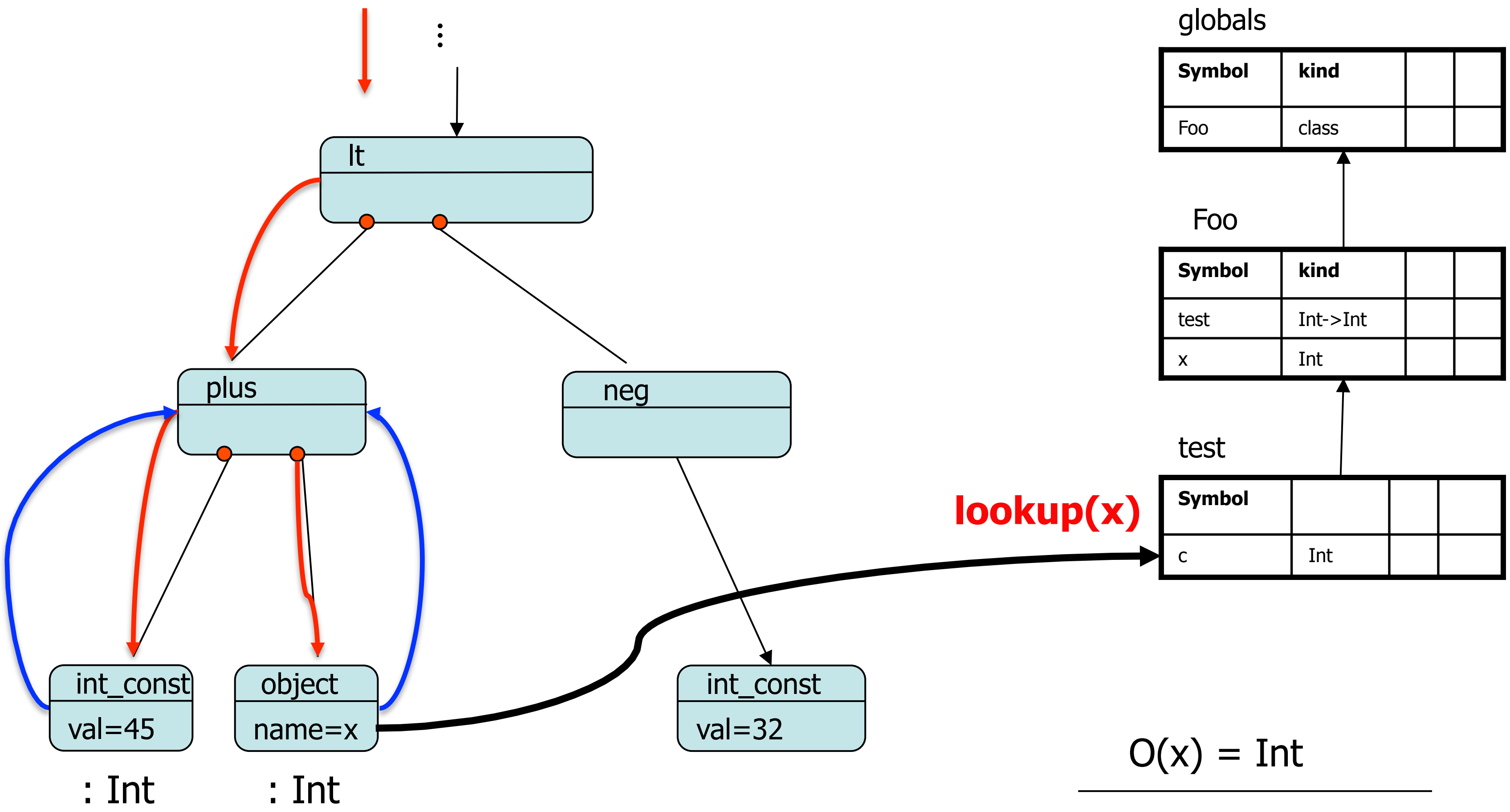
String String
“drive” + “drink”
string (in Java )
ERROR in Pascal ?

Int String
12 + “Rumster”
ERROR

# Type Checking - Implementation

- Single traversal over AST
- Types passed up the tree
- Type environment passed down the tree

# Example

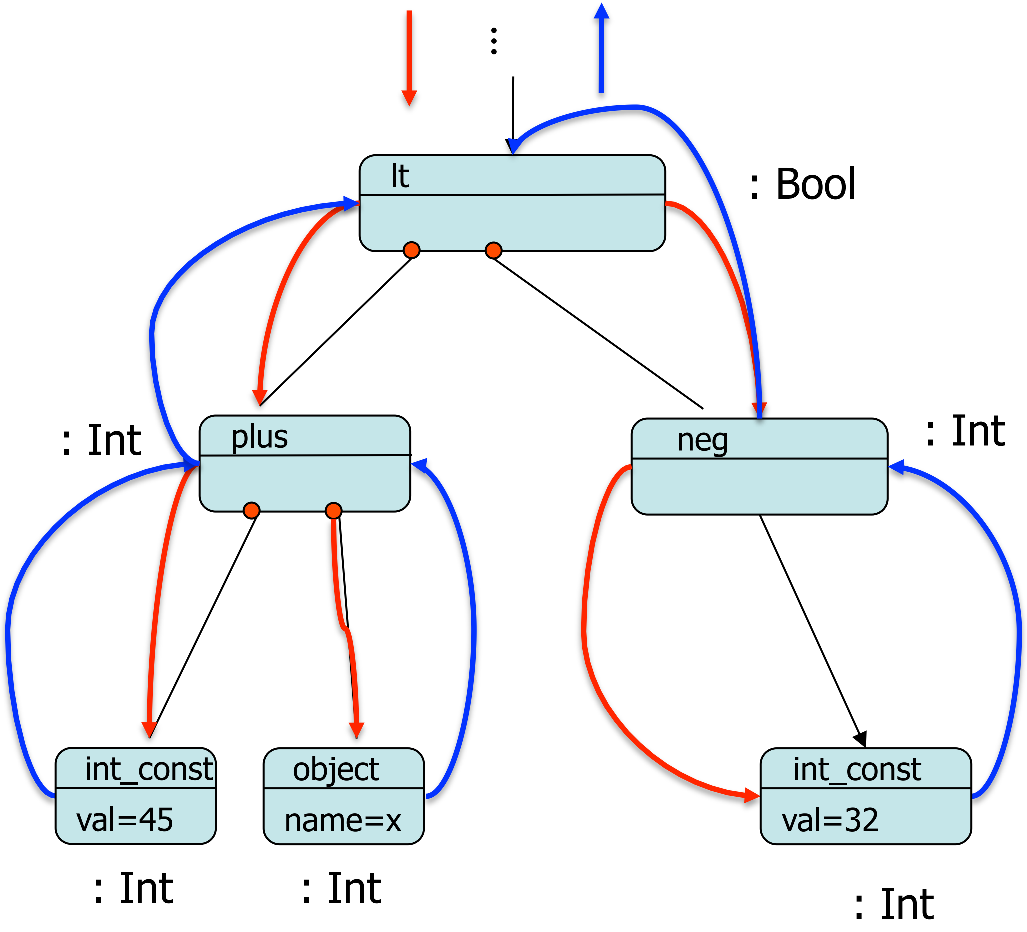



(45 + x) < (—32)

$$\frac{O(x) = \mathrm{Int}}{x : \mathrm{Int}}$$

$$\frac{}{\mathit{int\_const} : \mathrm{Int}}$$

# Example




(45 + x) < (—32)

$$\frac{E1 : Int \qquad E2 : Int}{E1 < E2 : Bool}$$

$$\frac{E1 : Int}{-E1 : Int}$$

$$\frac{E1 : Int \qquad E2 : Int}{E1 + E2 : Int}$$

$$\frac{O(x) = Int}{x : Int}$$

$$\frac{}{\mathit{int\_const} : Int}$$

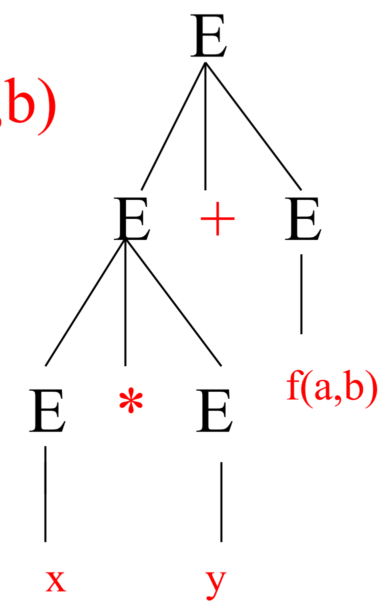
# Dynamic and Static Types

- The dynamic type of an object is the class that is used in the new expression
  - a runtime notion
  - even languages that are statically typed have dynamic types
- The static type of an expression captures all the dynamic types that the expression could have
  - a compile-time notion

# 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)



# 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)

# 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 → D ; D | id : T

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

S → S ; S | id := E | if E then S | while E do S

E → num | id | E + E | E [ E ] | E^ | E ( E )

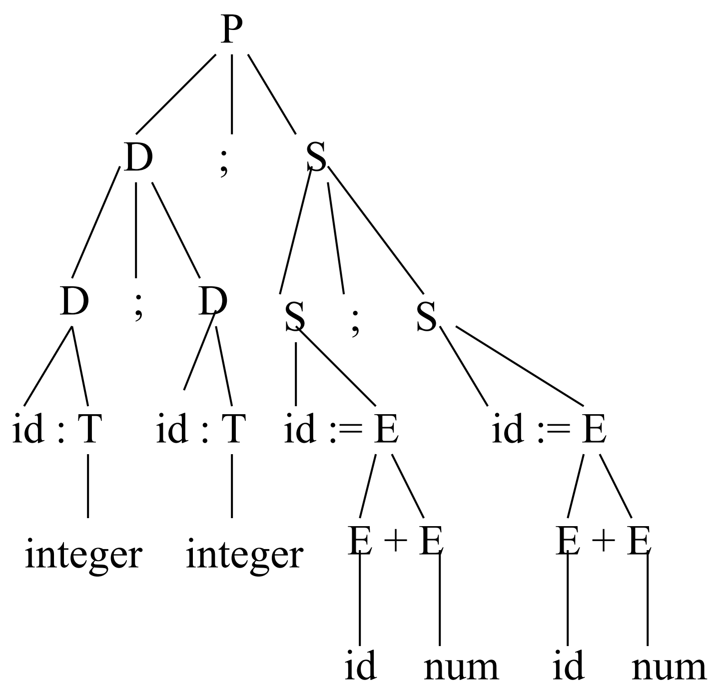
How can we typecheck strings in this language?

# Example of language:

i: integer; j: integer;

i := i + 1;

j := i + 1



# Processing Declarations

Put info into
the symbol table

$D \rightarrow D ; D$
$D \rightarrow id : T$
$T \rightarrow integer$
$T \rightarrow array [ num ] of T_1$
$T \rightarrow \hat{} T_1$
$T \rightarrow T_1 \times T_2$
$T \rightarrow T_1 \rightarrow T_2$

{insert(id.name,T.type);}
{T.type = integer;}
{T.type = array($T_1$.type,num); }
{T.type = pointer($T_1$.type);}
{T.type = product($T_1$.type, $T_2$.type);}
{T.type = function($T_1$.type, $T_2$.type);}

Accumulate information about
the declared type

# Example

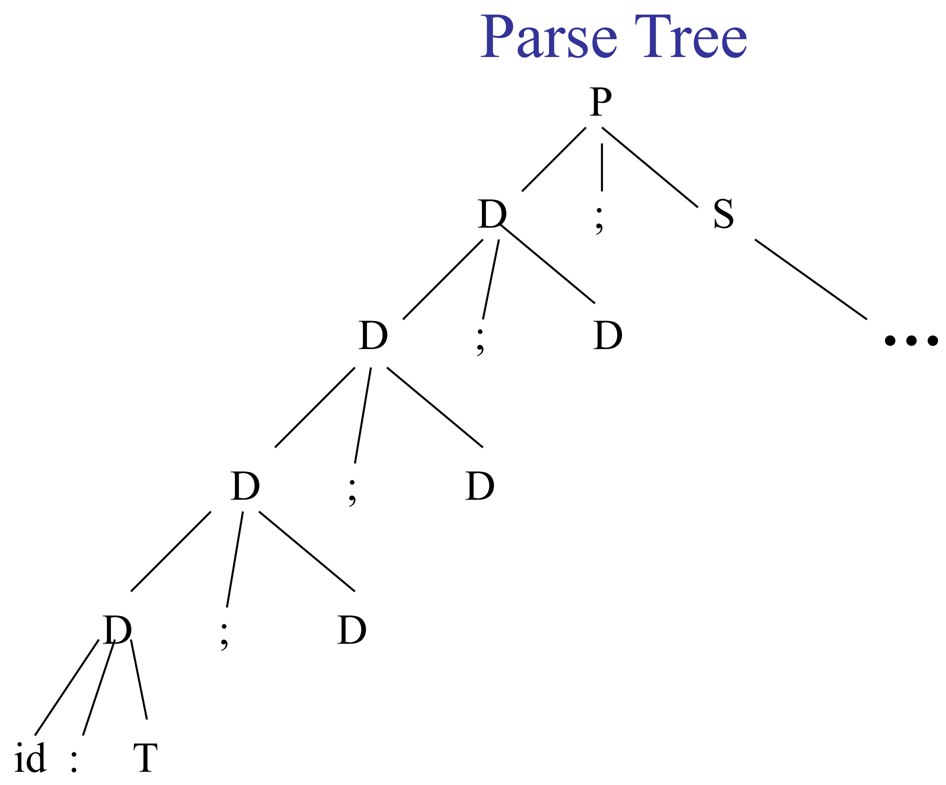
I: integer;

A: array[20] of integer;

B: array[20] of ^integer;

F: ^integer → integer;

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



# Example

I: integer;

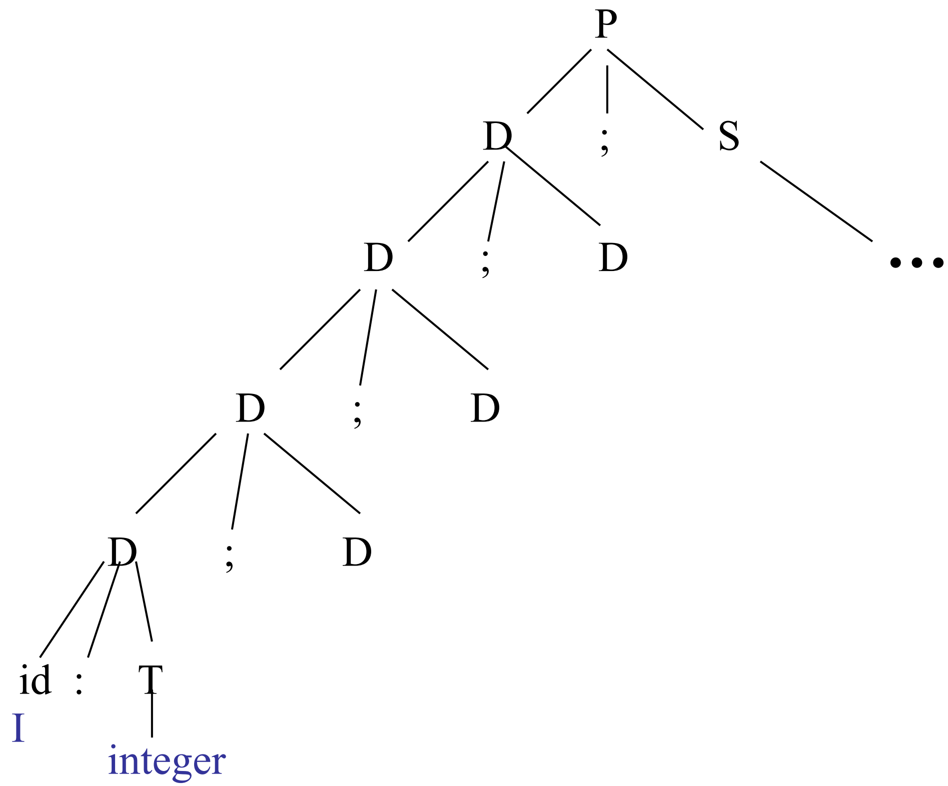
A: array[20] of integer;

B: array[20] of ^integer;

F: ^integer → integer;

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

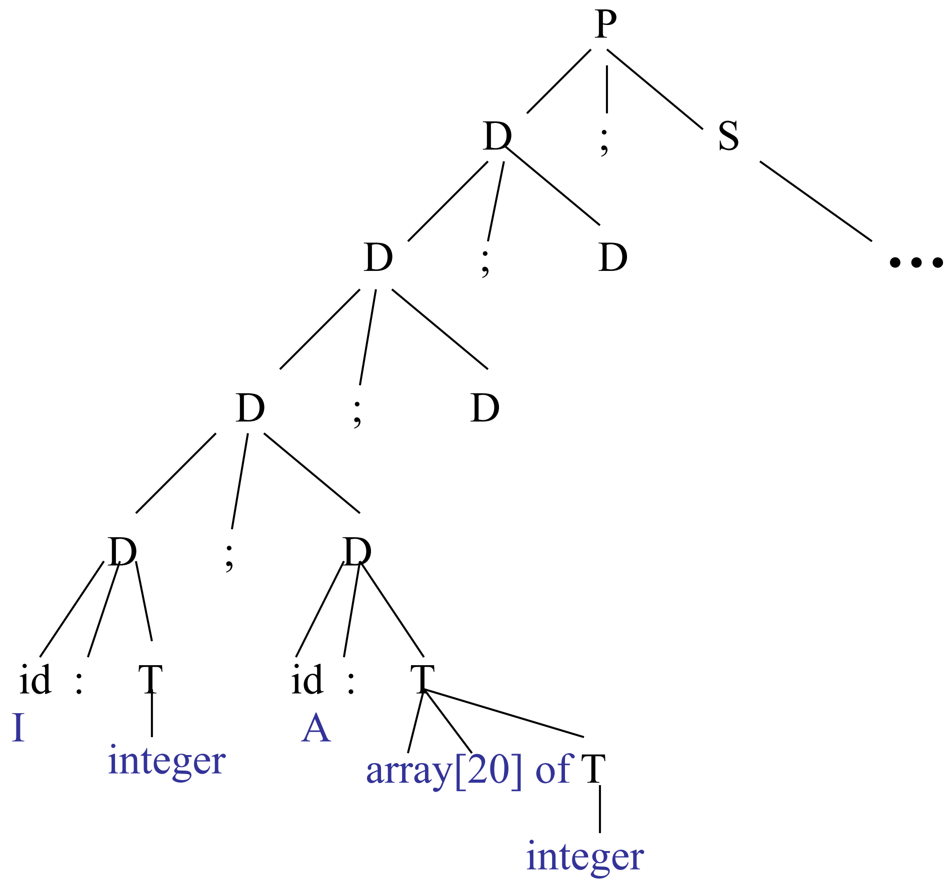
Parse Tree



# 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;

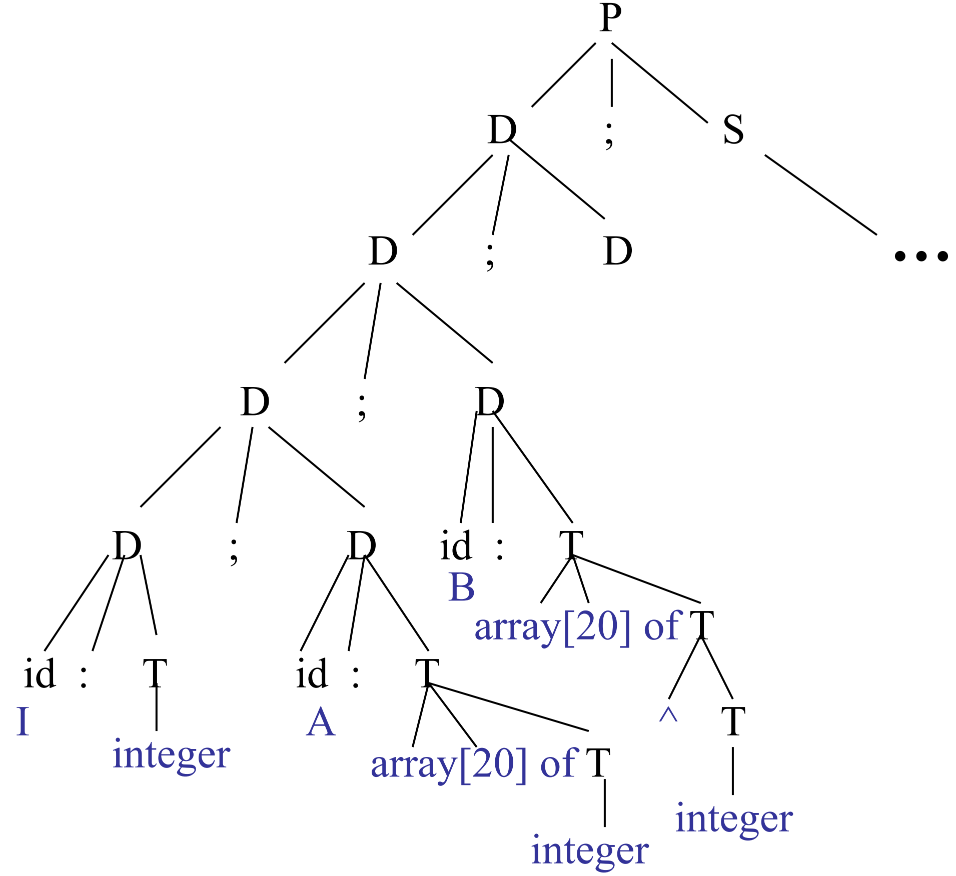
A: array[20] of integer;

B: array[20] of ^integer;

F: ^integer → integer;

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

Parse Tree



# Example

I: integer;

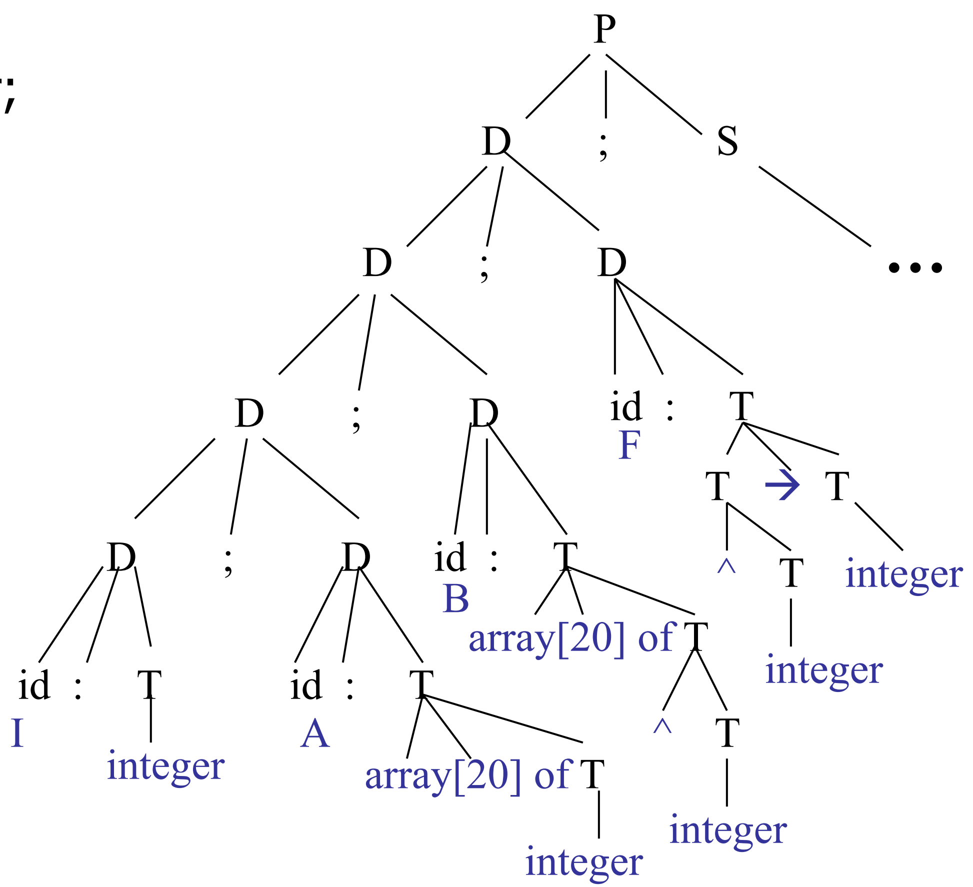
A: array[20] of integer;

B: array[20] of ^integer;

F: ^integer → integer;

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

Parse Tree



# Type Systems

- A **type system** is a collection of rules for assigning type expressions to various parts of the program.
- Why do we care so much about types?
  - Impose constraints that help enforce correctness.

# 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.

# Untyped languages

Single type that contains all values

- Example:
  - 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

# Safety

- Costs in terms of performance at both compile time and run time
- Different languages make different tradeoffs

| | Typed Languages | Untyped Languages |
|---|---|---|
| Safe | ML | Lisp |
| Unsafe | C | Assembly |

# 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

- If A and B match because they are both boolean ➔ Structural
- If A and B don't match because they have different name ➔ Name

# Implementing Structural Equivalence

- To determine whether two types are structurally equivalent, traverse the Trees:

```
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;
}
```

# 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)

# 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
- Polymorphism