# Type Checking

## Course : 2CS701 – Compiler Construction

Prof Monika Shah

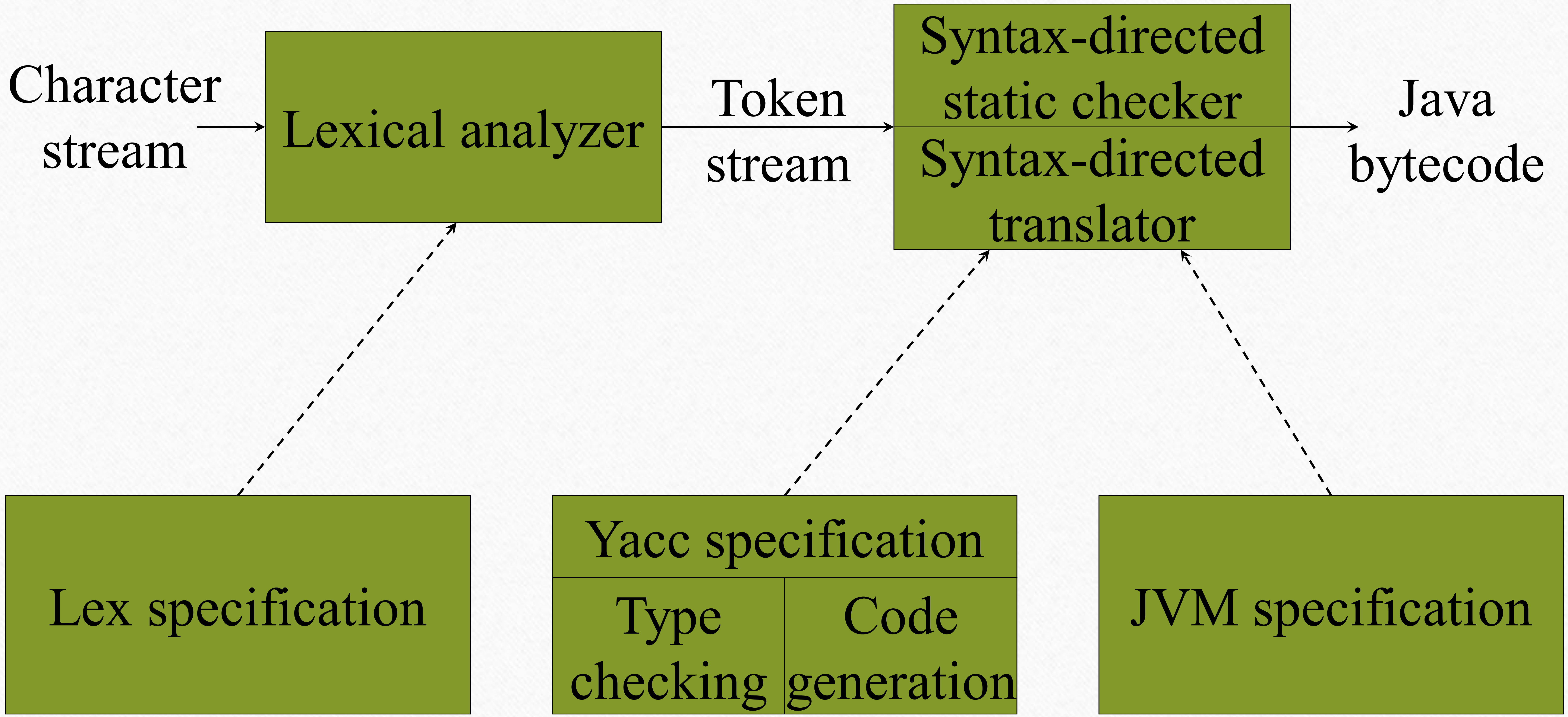
Nirma University

Ref : Ch.6 Compilers Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi, and Jeffrey Ullman

# Glimpse

- Review of Analysis phases
- Static Checking vs Dynamic Checking
- Types of Static Checking
- Type Expression
  - Name equivalence
  - Structure equivalence
- Type System
  - Declaration
  - Checking statements
  - Expression

# The Structure of our Compiler Revisited



# Review of Analysis phases

Check Source Program for

- Lexical Analyzer : Check Lexeme     Static Check
- Syntax Analyzer : Check Syntax     Static Check
- Semantic Analyzer : Check Semantic     Static Check     Dynamic Check
    - Type Check
    - Flow – of –control
    - Uniqueness check
    - Name related check

# Static checking vs Dynamic checking

Static Checking

- Check at compile time
- Check source program's properties at compile time
  - Type checking
  - Uniqueness
  - Flow-of-control
  - name related check

Dynamic Checking

- Check at runtime
- Check dynamic semantics during execution of target program
  - Type checking

# Static Checking

- Typical examples of static checking are
  - Type checks
  - Flow-of-control checks
  - Uniqueness checks
  - Name-related checks

# Type Checks, Overloading, Coercion, and Polymorphism

```
int op(int), op(float);
int f(float);
int a, c[10], d;

d = c + d;        // FAIL

*d = a;      // FAIL

a = op(d);        // OK: overloading (C++)

a = f(d);         // OK: coercion(implicit type casting)
                           of d to float

vector<int> v; // OK: template instantiation
```

# Flow-of-Control Checks

```
myfunc()
{ …
  break; // ERROR
}
```

```
myfunc()
{ …
  while (n)
  { …
    if (i>10)
      break; // OK
  }
}
```

```
myfunc()
{ …
  switch (a)
  { case 0:
      …
      break; // OK
    case 1:
      …
  }
}
```

# Uniqueness Checks

```
myfunc()
{ int i, j, i; // ERROR
   …
}
```

```
cnufym(int a, int a) // ERROR
{    …
}
```

```
struct myrec
{ int name;
};
struct myrec // ERROR
{ int id;
};
```

```
Symbol Table
===================
Variable Type
```

| Variable | Type |
|---|---|
| i | int |
| j | int |

Lexical Analyzer

Syntax Directed Type Check

# Name-Related Checks

```
LoopA: for (int I = 0; I < n; I++)
       { …
         if (a[I] == 0)
           break LoopB; // Java labeled loop
         …
       }
```

# One-Pass versus Multi-Pass Static Checking

- *One-pass compiler*: static checking in C, Pascal, Fortran, and many other languages is performed in one pass while intermediate code is generated
  - Influences design of a language: placement constraints

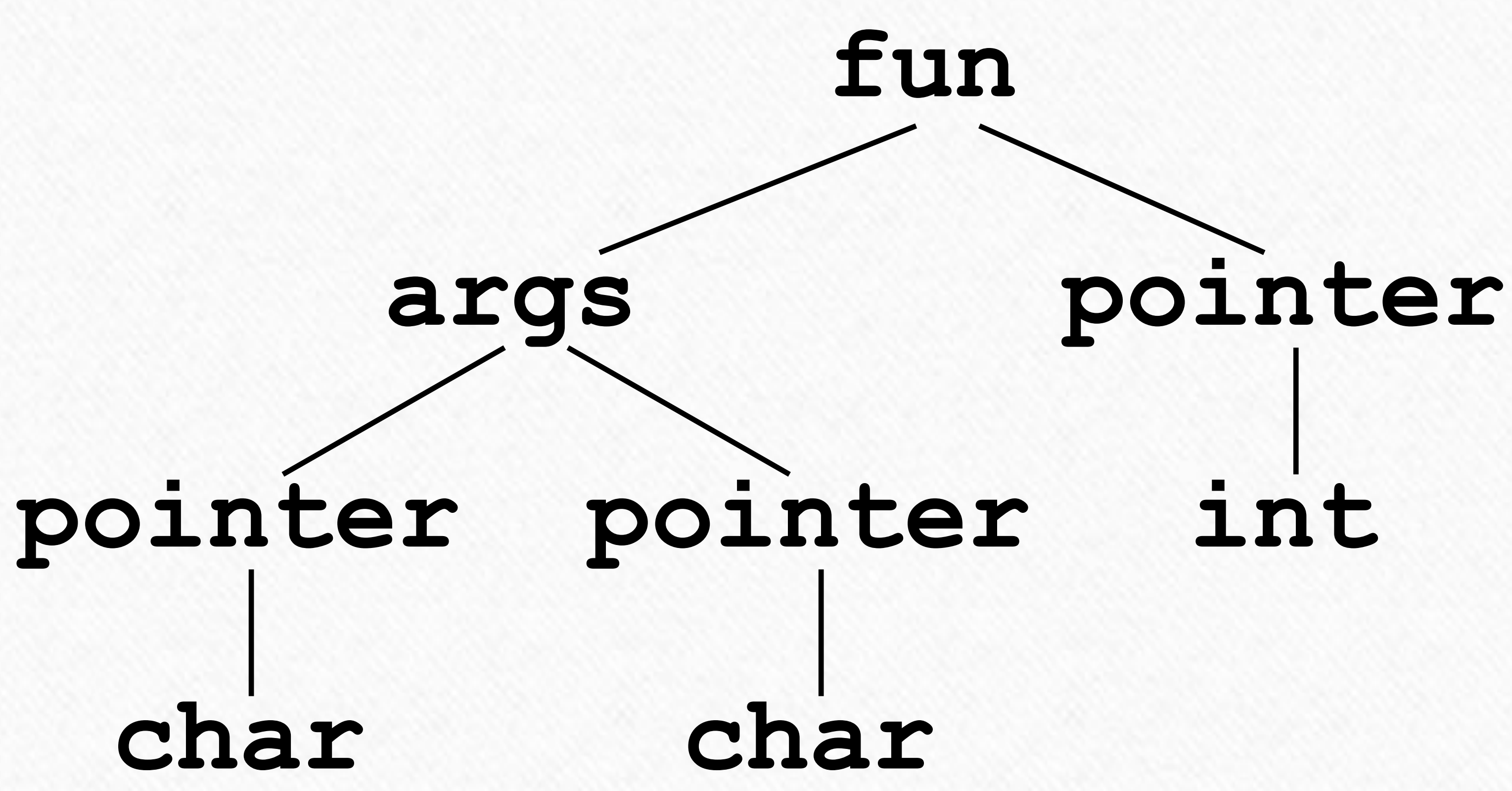
- *Multi-pass compiler*: static checking in Ada, Java, and C# is performed in a separate phase, sometimes by traversing a syntax tree multiple times

# Type Expressions

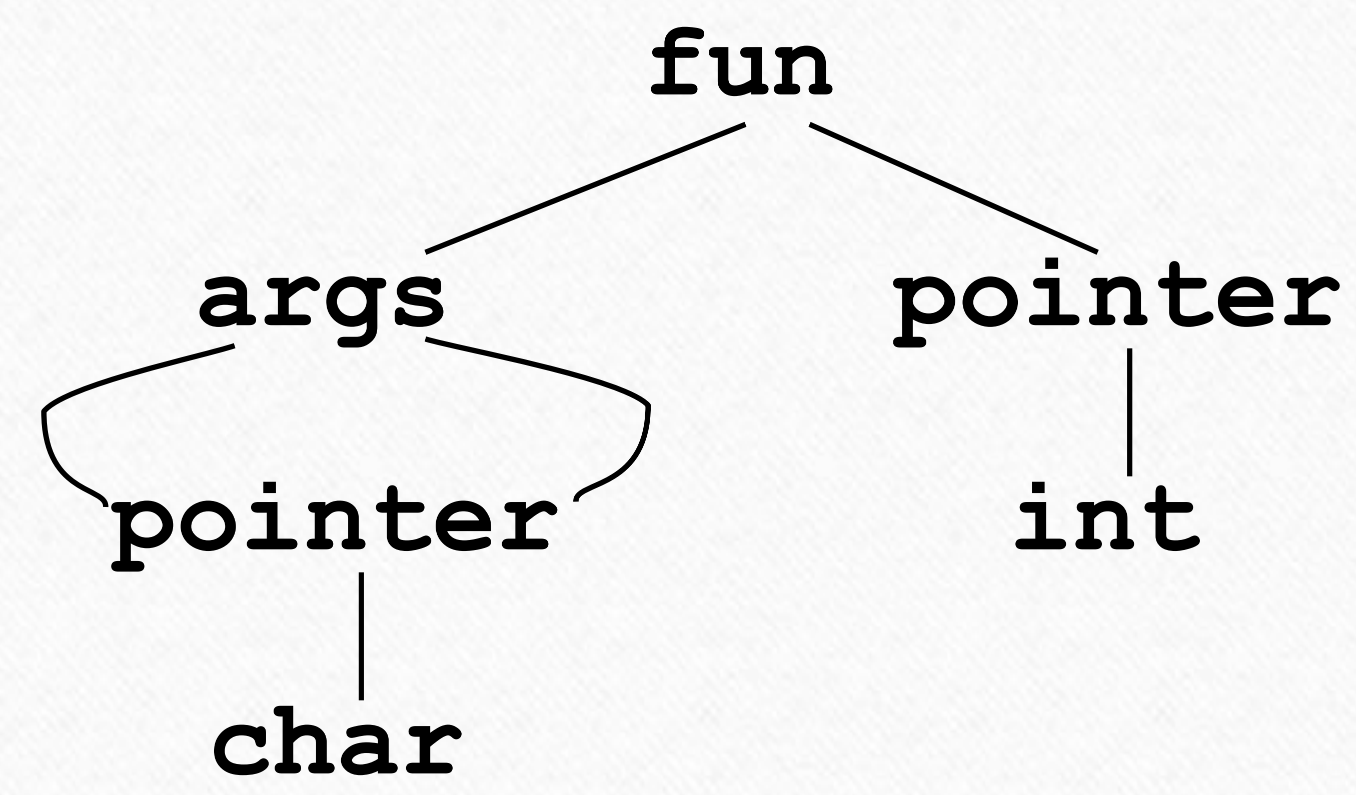
- *Type expressions* are used in declarations and type casts to define or refer to a type
  - *Primitive types*, such as **`int`** and **`bool`**
  - *Type constructors*, such as pointer-to, array-of, records and classes, templates, and functions
  - *Type names*, such as typedefs in C and named types in Pascal, refer to type expressions

# Graph Representations for Type Expressions

```
int *f(char*,char*)
```




Tree forms

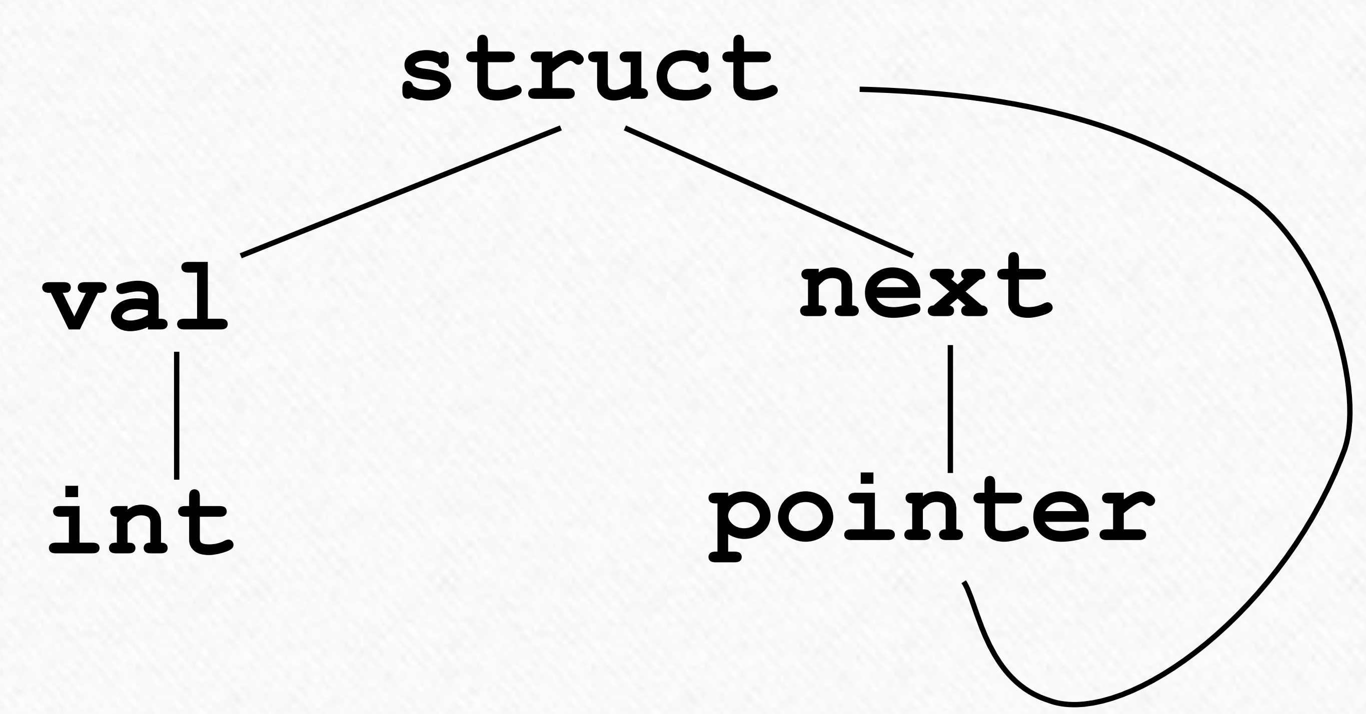



DAGs

# Cyclic Graph Representations

Source program

```
struct Node
{ int val;
  struct Node *next;
};
```




Internal compiler representation of the **Node** type: cyclic graph

# Name Equivalence

- Each *type name* is a distinct type, even when the type expressions the names refer to are the same
- Types are identical only if names match
- Used by Pascal (inconsistently)

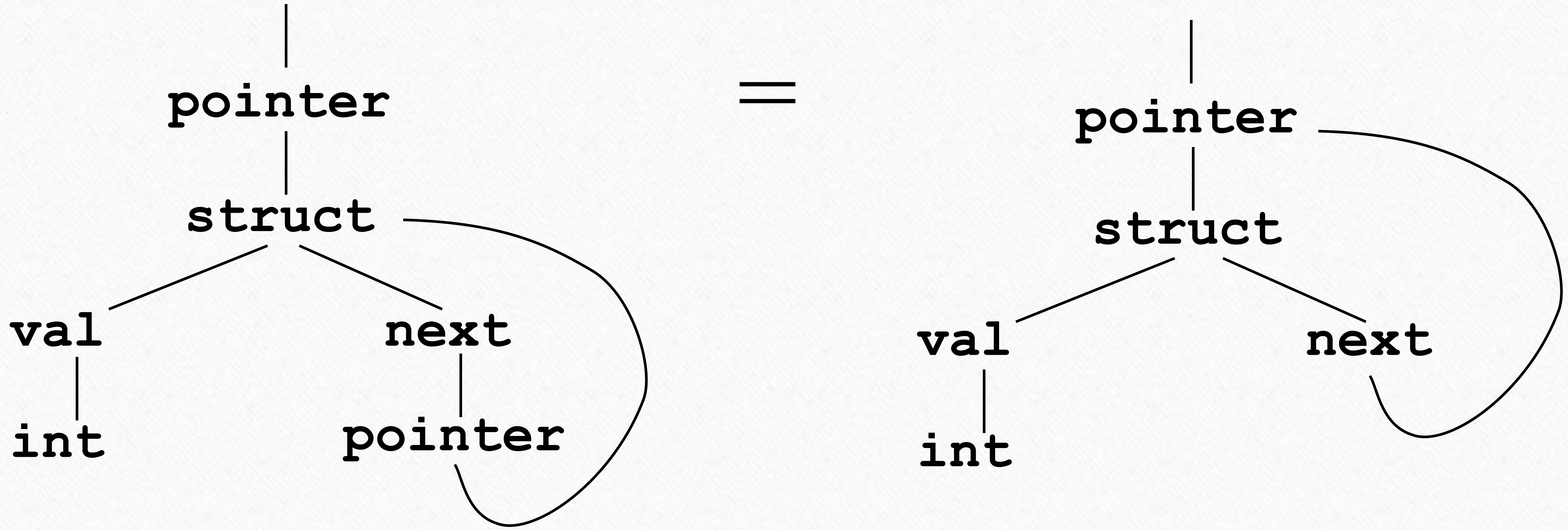
```
type link = ^node;
var next : link;
    last : link;
       p : ^node;
    q, r : ^node;
```

With name equivalence in Pascal:

```
p ≠ next
p ≠ last
p = q = r
next = last
```

# Structural Equivalence of Type Expressions

- Two types are the same if they are *structurally identical*
- Used in C, Java, C#

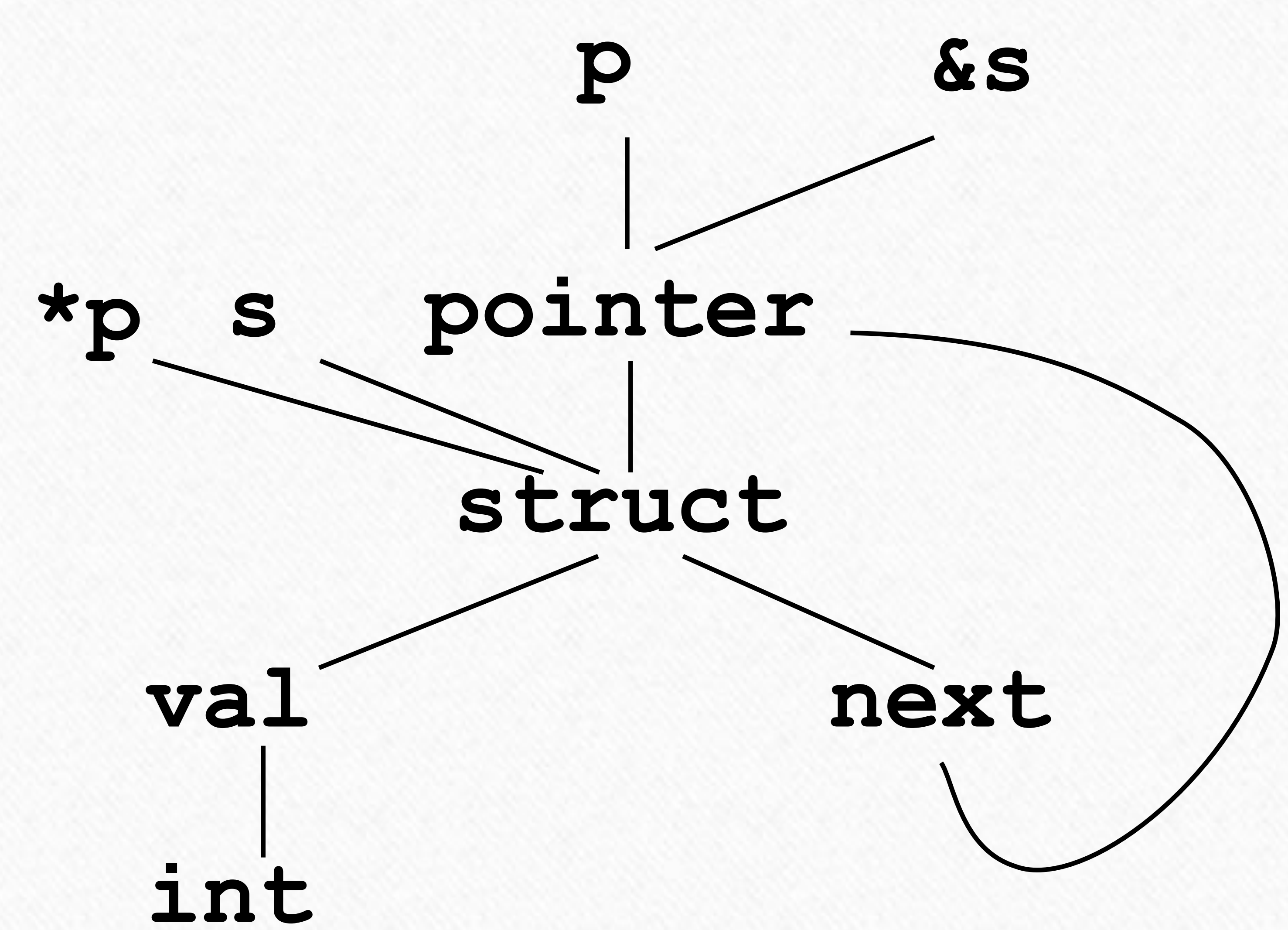


# Structural Equivalence of Type Expressions (cont'd)

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```
struct Node
{ int val;
  struct Node *next;
};
struct Node s, *p;

… p = &s; // OK
… *p = s; // OK
```



# Type Systems

- A *type system* defines a set of types and rules to assign types to programming language constructs
- Informal type system rules, for example "*if both operands of addition are of type integer, then the result is of type integer*"
- Formal type system rules: Post system

# A Simple Language Example

$P \to D \; \mathbf{;} \; S$

$D \to D \; \mathbf{;} \; D$
$\quad\mid \mathbf{id} : T$

$T \to \mathbf{boolean}$
$\quad\mid \mathbf{char}$
$\quad\mid \mathbf{integer}$
$\quad\mid \mathbf{array} \; [ \; \mathbf{num} \; ] \; \mathbf{of} \; T$
$\quad\mid \mathbf{\hat{}} \; T$

$S \to \mathbf{id} := E$
$\quad\mid \mathbf{if} \; E \; \mathbf{then} \; S$
$\quad\mid \mathbf{while} \; E \; \mathbf{do} \; S$
$\quad\mid S \; \mathbf{;} \; S$

$E \to \mathbf{true}$
$\quad\mid \mathbf{false}$
$\quad\mid \mathbf{literal}$
$\quad\mid \mathbf{num}$
$\quad\mid \mathbf{id}$
$\quad\mid E \; \mathbf{and} \; E$
$\quad\mid E + E$
$\quad\mid E \; [ \; E \; ]$
$\quad\mid E \; \mathbf{\hat{}}$

Pointer to $T$

Pascal-like pointer dereference operator

# Simple Language Example: Declarations

$D \rightarrow$ **id** : $T$ { *addtype*(**id**.entry, $T$.type) }
$T \rightarrow$ **boolean** { $T$.type := *boolean* }
$T \rightarrow$ **char** { $T$.type := *char* }
$T \rightarrow$ **integer** { $T$.type := *integer* }
$T \rightarrow$ **array [ num ] of** $T_1$ { $T$.type := *array*(1..**num**.val, $T_1$.type) }
$T \rightarrow$ ^ $T_1$ { $T$.type := *pointer*($T_1$)

Parametric types:
type constructor

# Simple Language Example: Checking Statements

$S \rightarrow$ **id** **:=** $E$ { $S$.type := **if** **id**.type = $E$.type **then** *void* **else** *type_error* }

Note: the type of **id** is determined by scope's environment:
**id**.type = *lookup*(**id**.entry)

# Simple Language Example: Checking Statements (cont'd)

$$S \rightarrow \textbf{if } E \textbf{ then } S_1 \quad \{\ S.\text{type} := \textbf{if } E.\text{type} = \textit{boolean}$$
$$\textbf{then } S_1.\text{type}$$
$$\textbf{else } \textit{type\_error}\ \}$$

$$S \rightarrow \textbf{while } E \textbf{ do } S_1 \quad \{\ S.\text{type} := \textbf{if } E.\text{type} = \textit{boolean} \textbf{ then } S_1.\text{type}$$
$$\textbf{else } \textit{type\_error}\ \}$$

$$S \rightarrow S_1 \textbf{ ; } S_2 \{\ S.\text{type} := \textbf{if } S_1.\text{type} = \textit{void} \textbf{ and } S_2.\text{type} = \textit{void} \textbf{ then } \textit{void}$$
$$\textbf{else } \textit{type\_error}\ \}$$

# Simple Language Example: Checking Expressions

$E \rightarrow$ **true** $\{ E.\text{type} = \textit{boolean} \}$

$E \rightarrow$ **false** $\{ E.\text{type} = \textit{boolean} \}$

$E \rightarrow$ **literal** $\{ E.\text{type} = \textit{char} \}$

$E \rightarrow$ **num** $\{ E.\text{type} = \textit{integer} \}$

$E \rightarrow$ **id** $\{ E.\text{type} = \textit{lookup}(\textbf{id}.\text{entry}) \}$

…

# Simple Language Example: Checking Expressions (cont'd)

$E \rightarrow E_1 + E_2$ { $E$.type := **if** $E_1$.type = *integer* **and** $E_2$.type = *integer* **then** *integer* **else** *type_error* }

$E \rightarrow E_1$ **and** $E_2$ { $E$.type := **if** $E_1$.type = *boolean* **and** $E_2$.type = *boolean* **then** *boolean* **else** *type_error* }

$E \rightarrow E_1$ **[** $E_2$ **]** { $E$.type := **if** $E_1$.type = *array*($s$, $t$) **and** $E_2$.type = *integer* **then** $t$ **else** *type_error* }

$E \rightarrow *E_1$ { $E$.type := **if** $E_1$.type = *pointer*($t$) **then** $t$ **else** *type_error* }

# A Simple Language Example: Functions

$T \rightarrow T$ **->** $T$

Function type declaration

$E \rightarrow E$ **(** $E$ **)**

Function call

Example:

```
v : integer;
odd : integer -> boolean;
if odd(3) then
    v := 1;
```

# Simple Language Example: Function Declarations

$T \rightarrow T_1$ **->** $T_2$ { $T$.type := *function*($T_1$.type, $T_2$.type) }

Parametric type:
type constructor

# Self Evaluation

**Perform following semantic check using syntax directed translation**

- Type checking of following construct in C
  - for statement, where expression are not compulsory
  - Switch statement
- Flow of control
  - Continue statement in ‘C’ Lanaguage
- Uniqueness check for case label in ‘C’

# Next Session Topic

- Structural Equivalence Algorithm
- Coercion

- Define Type system using YACC