# CS 4300: Compiler Theory

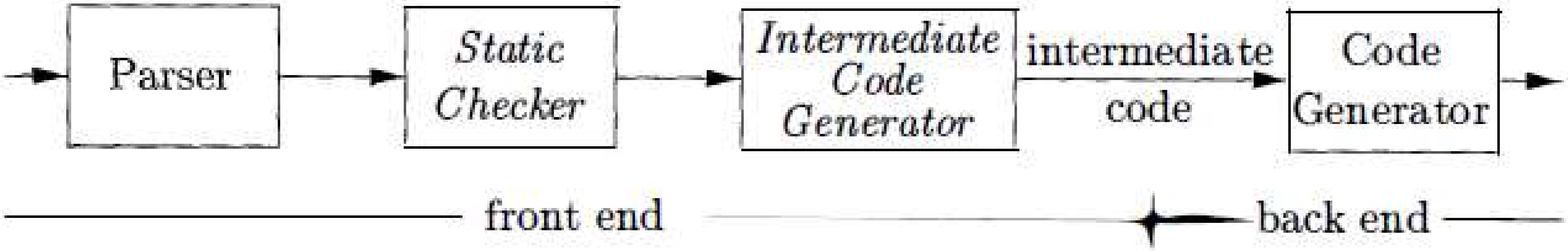
# Chapter 6
# Intermediate-Code Generation

*Xuejun Liang*

*2019 Fall*

# Introduction

Logical structure of a compiler front end



A sequence of intermediate representations

Syntax trees are high level

Three-address code can range from high-level to low-level, depending on the choice of operators

# Static versus Dynamic Checking

- Static checking: checked at compile time
  - Compiler enforces programming language's static semantics
  - Typical examples of static checking:
    - Type checks
    - Flow-of-control checks
    - Uniqueness checks
    - Name-related checks
- Dynamic semantics: checked at run time
  - Compiler generates verification code to enforce programming language's dynamic semantics

# Type Checking, Overloading, Coercion, Polymorphism

```
class X { virtual int m(); } *x;
class Y: public X { virtual int m(); } *y;
int op(int), op(float);
int f(float);
int a, c[10], d;

d = c + d;          // FAIL
*d = a;             // FAIL
a = op(d);          // OK: static overloading (C++)
a = f(d);           // OK: coersion of d to float
a = x->m();         // OK: dynamic binding (C++)
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;
};
```

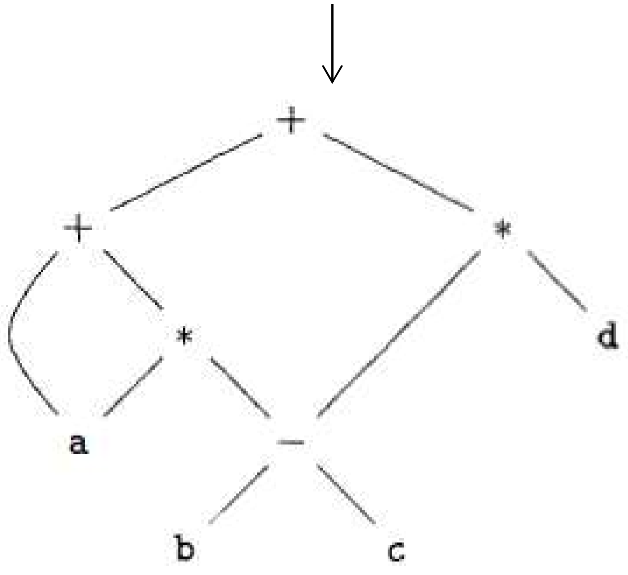
# Outlines (Sections)

1. Variants of Syntax Trees
2. Three-Address Code
3. Types and Declarations
4. Translation of Expressions
5. Type Checking
6. Control Flow
7. Backpatching
8. Switch-Statements
9. Intermediate Code for Procedures

# 1. Variants of Syntax Trees

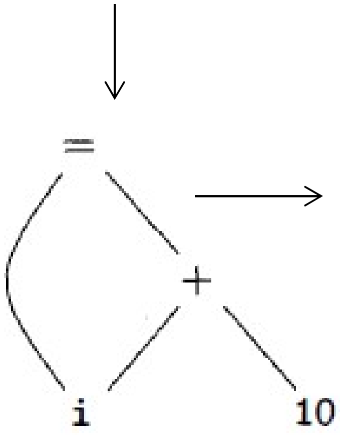
A directed acyclic graph (called a DAG) for an expression identifies the common subexpressions of the expression

a + a * (b - c) + (b - c) * d



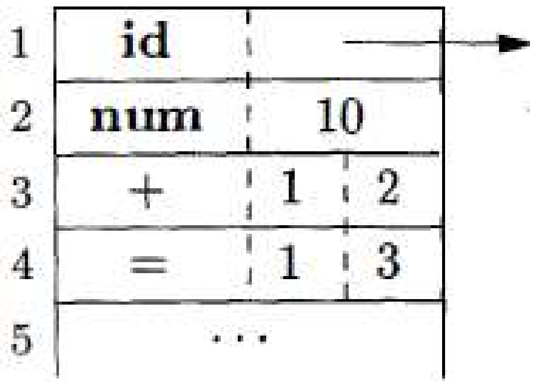
DAG

i = i + 10



DAG

to symbol table



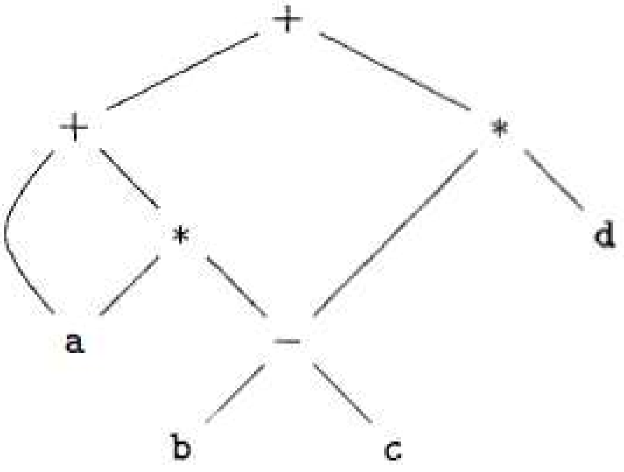
| | | | |
|---|---|---|---|
| 1 | id | → | |
| 2 | num | 10 | |
| 3 | + | 1 | 2 |
| 4 | = | 1 | 3 |
| 5 | . . . | | |

Array

Value number

# 2. Three-Address Code

In three-address code, there is at most one operator on the right side of an instruction. An address can be: name, constant, compiler-generated temporary.



DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

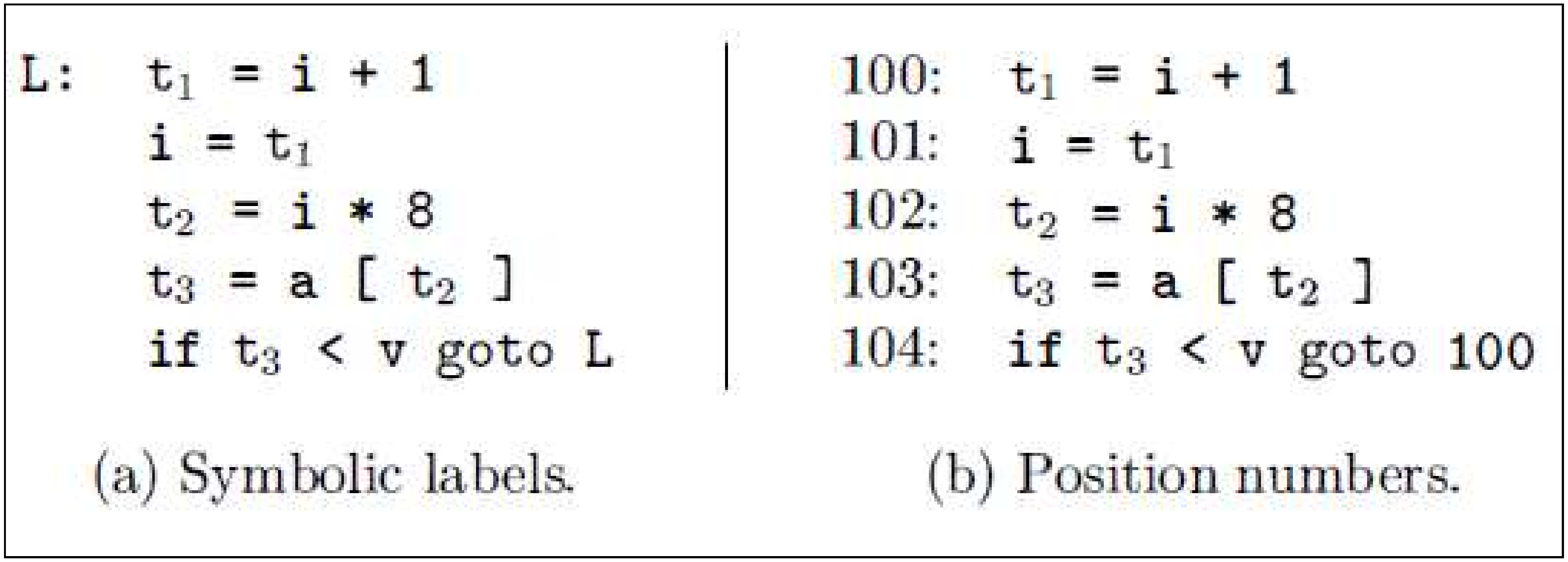
Three-address code

# Common Three-Address Instructions

| | |
|---|---|
| 1. Assignment instruction | $x = y \; op \; z$ |
| 2. Assignment | $x = op \; y$ |
| 3. Copy instruction | $x = y$ |
| 4. Indexed copy instruction | $x = y[i]$ and $x[i] = y$ |
| 5. Address and pointer assignment: | $x = \&y$, $x = *y$, and $*x = y$ |
| 6. Unconditional jump | *goto L* |
| 7. Conditional jump | *if x relop y goto L* |
| 8. Conditional jump | *if x goto L* and *ifFalse x goto L* |
| 9. Procedure call $p(x_1, x_2, \ldots, x_n)$: | *param* $x_1$<br>*param* $x_2$<br>… …<br>*param* $x_n$<br>*call p, n* |

# Two Ways of Assigning Labels to Three-Address Statements

do i = i+ l; while (a[i] < v) ;



L: $t_1$ = i + 1
$\quad$ i = $t_1$
$\quad$ $t_2$ = i * 8
$\quad$ $t_3$ = a [ $t_2$ ]
$\quad$ if $t_3$ < v goto L

(a) Symbolic labels.

100: $t_1$ = i + 1
101: i = $t_1$
102: $t_2$ = i * 8
103: $t_3$ = a [ $t_2$ ]
104: if $t_3$ < v goto 100

(b) Position numbers.

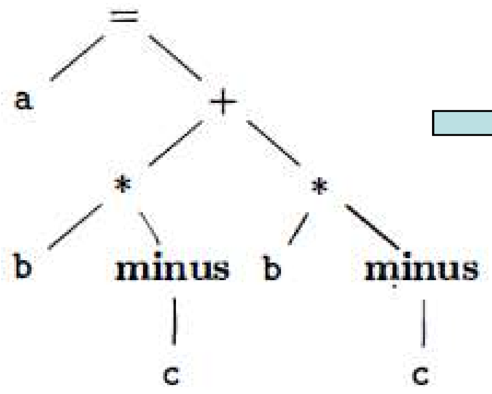
# Quadruples, Triples, and Indirect Triples

$t_1$ = minus c
$t_2$ = b * $t_1$
$t_3$ = minus c
$t_4$ = b * $t_3$
$t_5$ = $t_2$ + $t_4$
a = $t_5$

Three-address code

| | *op* | *arg*$_1$ | *arg*$_2$ | *result* |
|---|---|---|---|---|
| 0 | minus | c | | $t_1$ |
| 1 | * | b | $t_1$ | $t_2$ |
| 2 | minus | c | | $t_3$ |
| 3 | * | b | $t_3$ | $t_4$ |
| 4 | + | $t_2$ | $t_4$ | $t_5$ |
| 5 | = | $t_5$ | | a |
| | | … | | |

Quadruples



Syntax tree

| | *op* | *arg*$_1$ | *arg*$_2$ |
|---|---|---|---|
| 0 | minus | c | |
| 1 | * | b | (0) |
| 2 | minus | c | |
| 3 | * | b | (2) |
| 4 | + | (1) | (3) |
| 5 | = | a | (4) |
| | | … | |

Triples

| | *instruction* |
|---|---|
| 35 | (0) |
| 36 | (1) |
| 37 | (2) |
| 38 | (3) |
| 39 | (4) |
| 40 | (5) |
| | … |

+ Triples

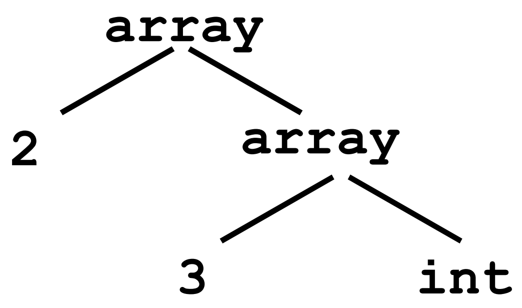
Indirect triples

# 3. Type Expressions

- A type expression is either a basic type or is formed by applying a type constructor to type expressions
  - Basic types: boolean, char, integer, float, etc.
  - Type constructors: pointer-to, array-of, records and classes, list-of, templates, and functions (s → t).
  - Type names: typedefs in C and named types in Pascal
- Type expressions may contain variables whose values are type expressions

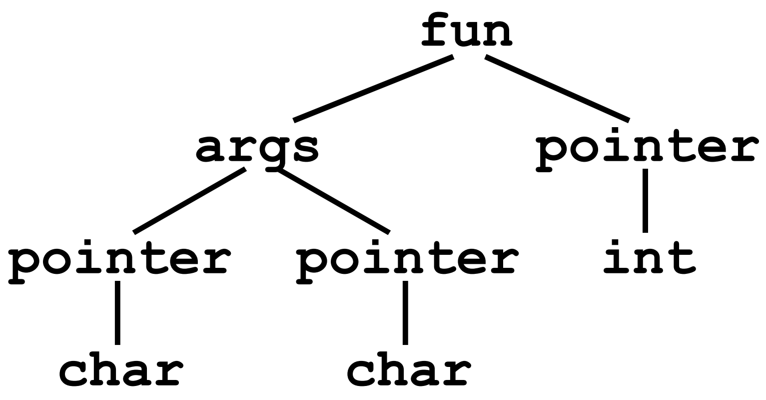
# Graph Representations for Type Expressions

`int [2][3]`

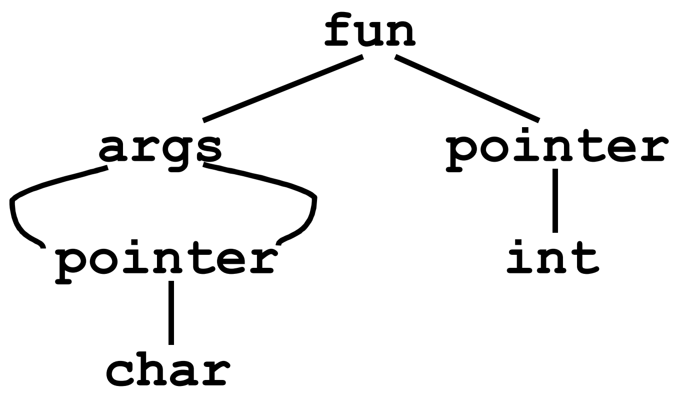


Tree

`int *fun(char*,char*)`



Tree



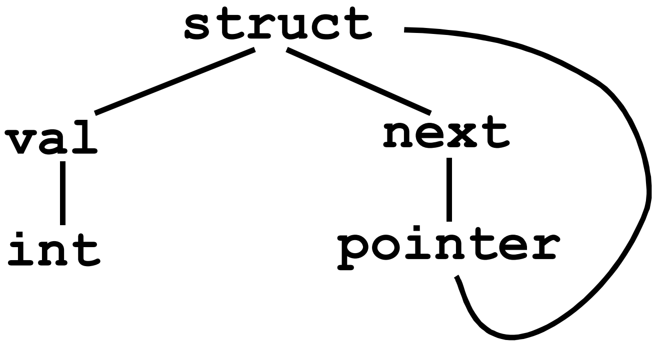
DAG

# Cyclic Graph Representations

Source program

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

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

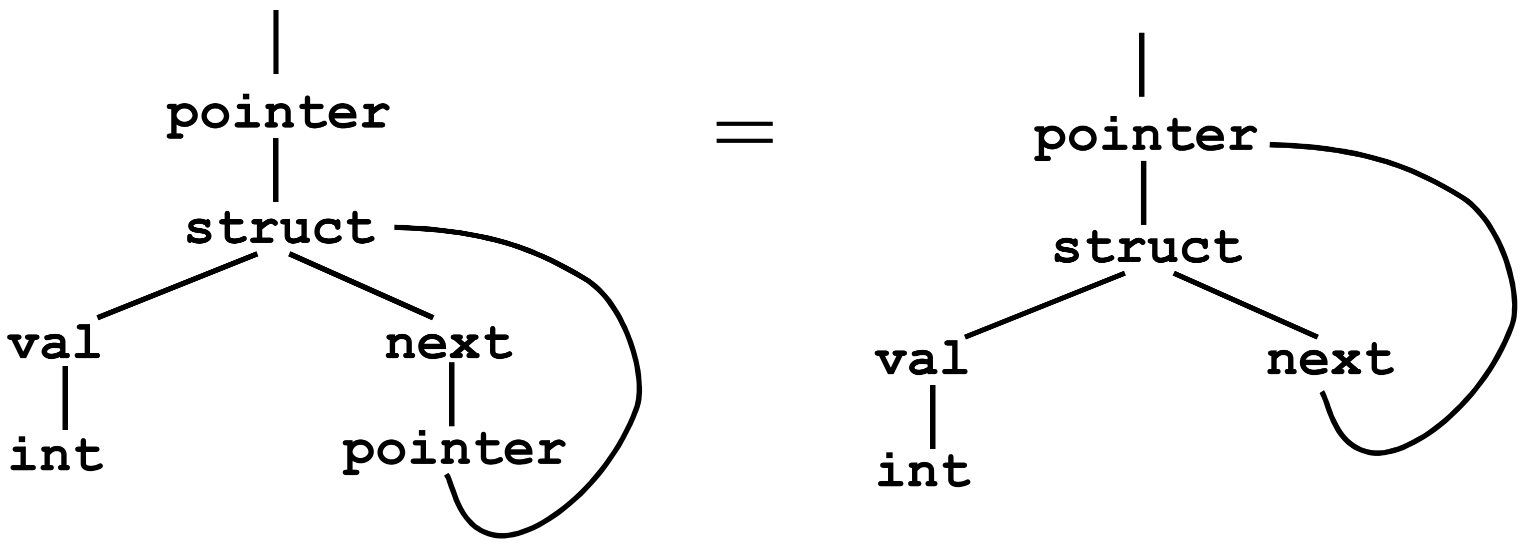


# Type Equivalence

- When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:
  - They are the same basic type.
  - They are formed by applying the same constructor to structurally equivalent types .
  - One is a type name that denotes the other .
- If type names are treated as standing for themselves, then the first two conditions in the above definition lead to name equivalence of type expressions

# Structural Equivalence Example

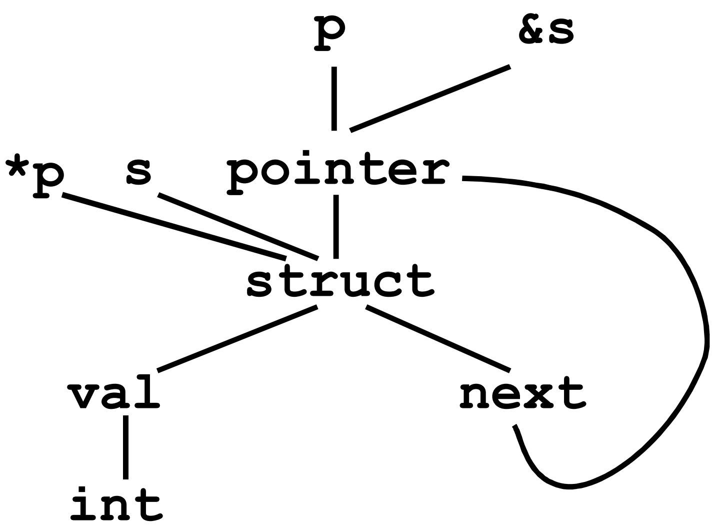
- Two types are the same if they are *structurally identical*
- Used in C/C++, Java, C#



# Type Equivalence Examples

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

struct Node s, *p;
p = &s; // OK
*p = s; // OK
p = s;  // ERROR
```

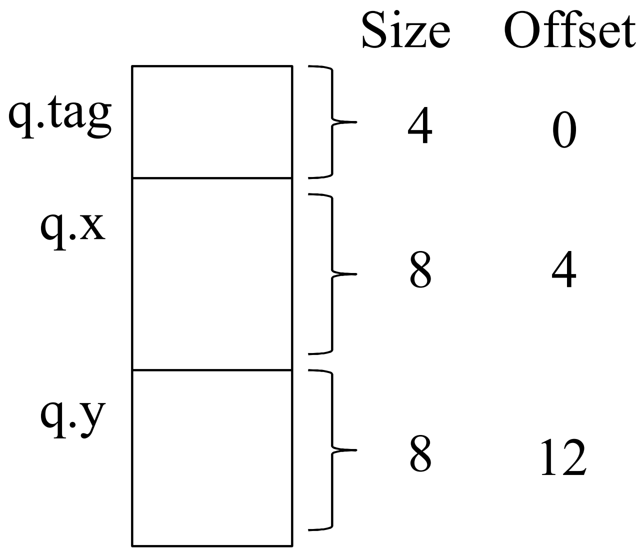


# Storage Layout for Local Names

Type Declarations

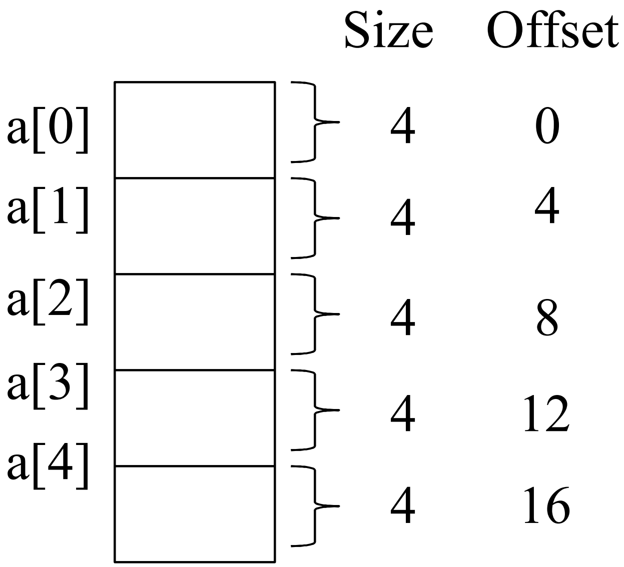
$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \textbf{record}\ \text{'\{'}\ D\ \text{'\}'} \\
B & \rightarrow & \textbf{int} \mid \textbf{float} \\
C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C
\end{array}
$$

```
record { int tag; float x; float y; } q;
```



| | Size | Offset |
|---|---|---|
| q.tag | 4 | 0 |
| q.x | 8 | 4 |
| q.y | 8 | 12 |

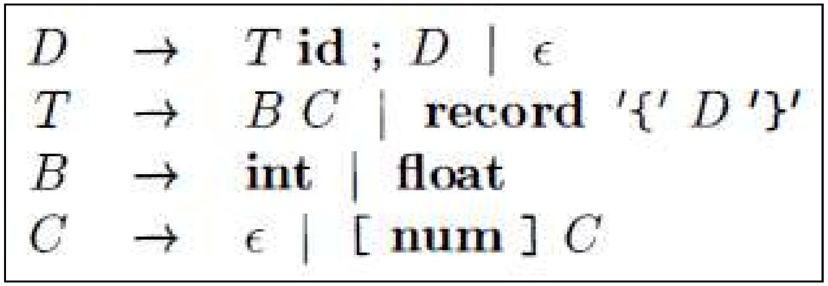
```
int [5] a;
```



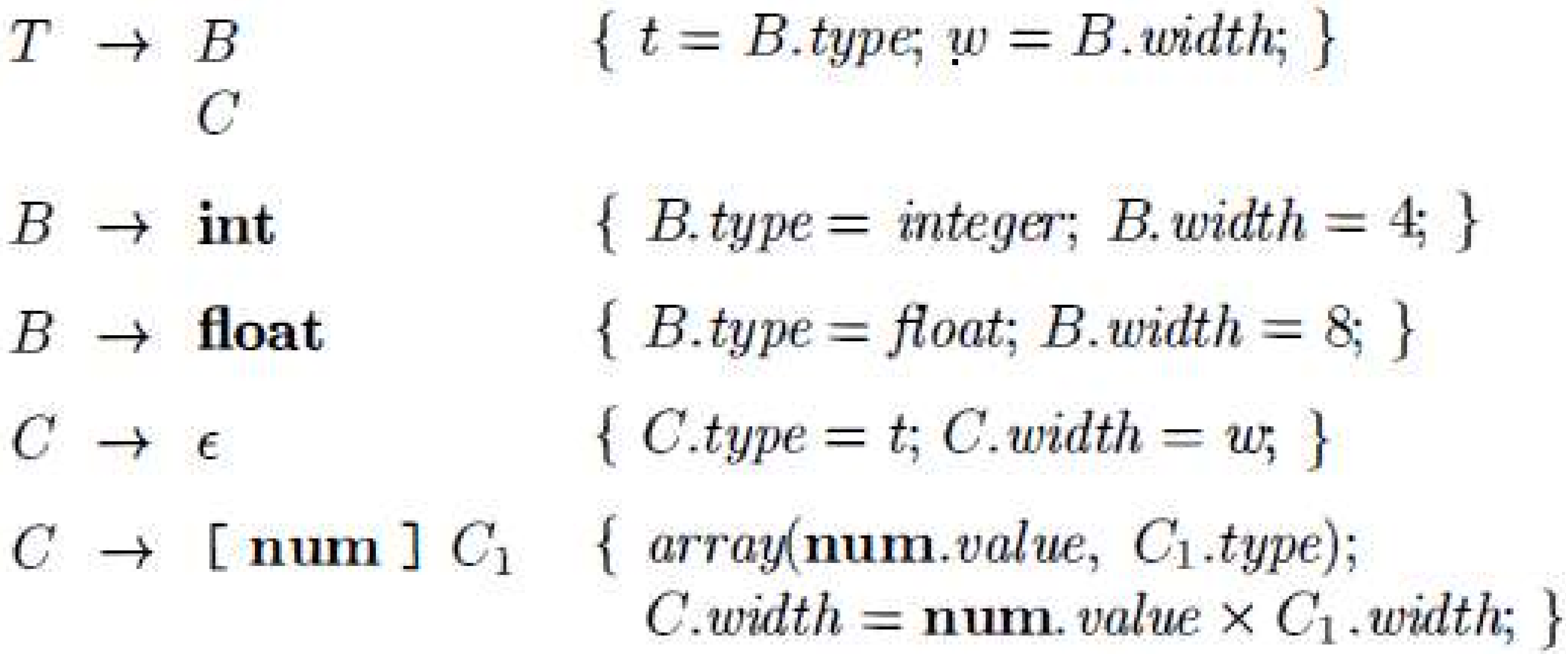
| | Size | Offset |
|---|---|---|
| a[0] | 4 | 0 |
| a[1] | 4 | 4 |
| a[2] | 4 | 8 |
| a[3] | 4 | 12 |
| a[4] | 4 | 16 |

# Computing Types and Their Widths

Type Declarations



$$
\begin{array}{lll}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ \text{'\{'}\ D\ \text{'\}'} \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$



| | |
|---|---|
| $T \rightarrow B$ | $\{\ t = B.type;\ w = B.width;\ \}$ |
| $\phantom{T \rightarrow{}} C$ | |
| $B \rightarrow \mathbf{int}$ | $\{\ B.type = integer;\ B.width = 4;\ \}$ |
| $B \rightarrow \mathbf{float}$ | $\{\ B.type = float;\ B.width = 8;\ \}$ |
| $C \rightarrow \epsilon$ | $\{\ C.type = t;\ C.width = w;\ \}$ |
| $C \rightarrow [\ \mathbf{num}\ ]\ C_1$ | $\{\ array(\mathbf{num}.value,\ C_1.type);$<br>$C.width = \mathbf{num}.value \times C_1.width;\ \}$ |

# Sequences of Declarations

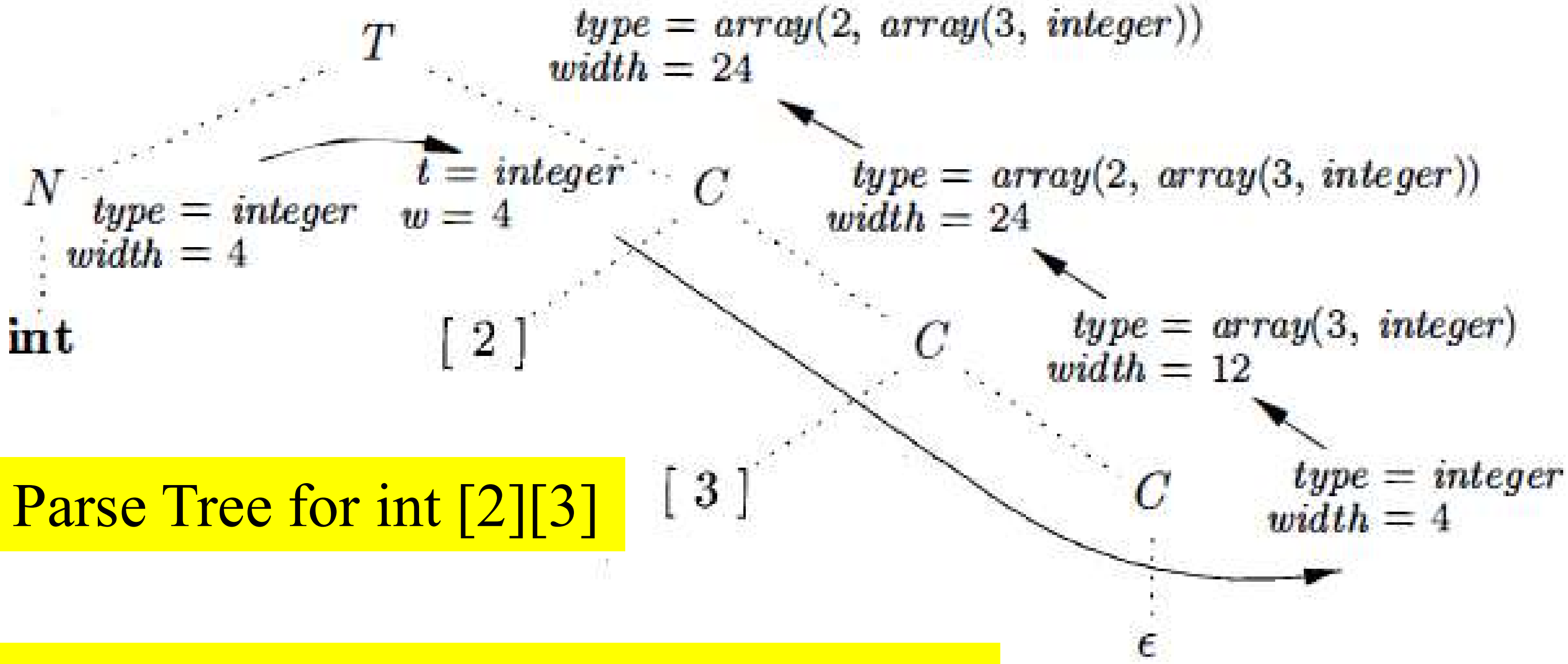
Computing the relative addresses of declared names

$$
\begin{array}{lcll}
P & \rightarrow & & \{\ offset = 0;\ \} \\
 & & D & \\
D & \rightarrow & T\ \mathbf{id}\ ; & \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset); \\
 & & & \quad offset\ =\ offset + T.width;\ \} \\
 & & D_1 & \\
D & \rightarrow & \epsilon &
\end{array}
$$

Handling of field names in records

$$
\begin{array}{lcll}
T & \rightarrow & \mathbf{record}\ '\{' & \{\ Env.push(top);\ top = \mathbf{new}\ Env(); \\
 & & & \quad Stack.push(offset);\ offset = 0;\ \} \\
 & & D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\
 & & & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

# Examples:



Parse Tree for int [2][3]

Determine types and relative addresses

float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;

# 4. Translation of Expressions

Example
a = b + - c

<table>
<tr><th>PRODUCTION</th><th>SEMANTIC RULES</th></tr>
<tr><td>S → id = E ;</td><td>S.code = E.code ||<br>gen( top.get(id.lexeme) '=' E.addr)</td></tr>
<tr><td>E → $E_1 + E_2$</td><td>E.addr = <b>new</b> Temp()<br>E.code = $E_1$.code || $E_2$.code ||<br>gen(E.addr '=' $E_1$.addr '+' $E_2$.addr)</td></tr>
<tr><td>| - $E_1$</td><td>E.addr = <b>new</b> Temp ()<br>E.code = $E_1$.code ||<br>gen(E.addr '=' 'minus' $E_1$.addr)</td></tr>
<tr><td>| ( $E_1$ )</td><td>E.addr = $E_1$.addr<br>E.code = $E_1$.code</td></tr>
<tr><td>| <b>id</b></td><td>E.addr = top.get(id.lexeme)<br>E.code = ' '</td></tr>
</table>

$t_1$ = **minus** c
$t_2 = b + t_1$
$a = t_2$

Figure 6.19: Three-address code for expressions

# Translation of Expressions (cont.)

| | |
|---|---|
| S → id = E ; | { gen( top.get(id.lexeme) '=' E.addr) ; } |
| E → $E_1$ + $E_2$ | { E.addr = new Temp();<br>gen(E.addr '=' $E_1$.addr '+' $E_2$.addr) ; } |
| \| - $E_1$ | { E.addr = new Temp() ;<br>gen(E.addr '=' 'minus' $E_1$.addr) ; } |
| \| ( $E_1$ ) | { E.addr = $E_1$.addr; } |
| \| **id** | { E.addr = top.get(id.lexeme) ;} |

**Figure 6.20: Generating three-address code for expressions incrementally**

In the incremental approach, gen not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far.

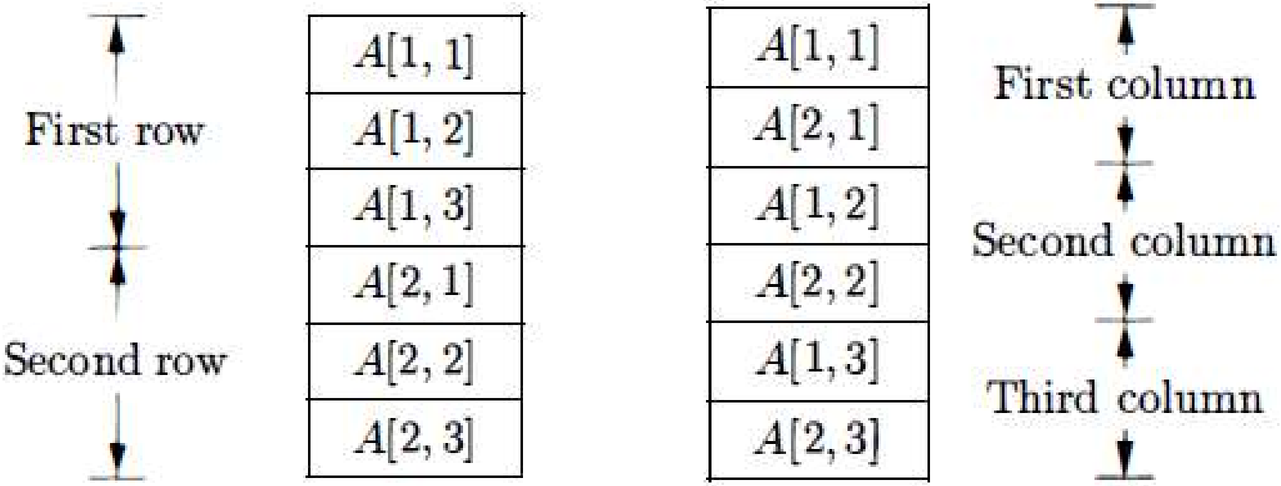
# Addressing Array Elements

$\mathbf{a[i].addr = base + i \times w}$

$\mathbf{A[i_l][i_2].addr = base + i_l \times w_l + i_2 \times w_2}$

$\mathbf{A[i_l][i_2]\ldots[i_k].addr = base + i_l \times w_l + i_2 \times w_{2.}\ ...\ + i_k \times w_k} \quad \mathbf{(6.4)}$

Layouts for a two-dimensional array



(a) Row Major

(b) Column Major

# Translation of Array References

| | |
|---|---|
| S → L = E | { gen(L.addr.base '[' L.addr ']' '=' E.addr); } |
| E → L | { E.addr = new Temp();<br>gen(E.addr '=' L.array.base '[' L.addr ' ] '); } |
| L → **id** [ E ] | { L.array = top.get(id.lexeme) ;<br>L.type = L.array.type.elem;<br>L.addr = new Temp();<br>gen(L.addr '=' E.addr '*' L.type.width); } |
| \| $L_1$ [ E ] | { L.array = $L_1$.array;<br>L.type = $L_1$.type.elem;<br>t = **new** Temp();<br>L.addr = **new** Temp();<br>gen(t '=' E.addr '*' L.type.width);<br>gen(L.addr '=' $L_1$.addr '+' t); } |

Figure 6.22: Semantic actions for array references

# Translation of Array References (Cont.)

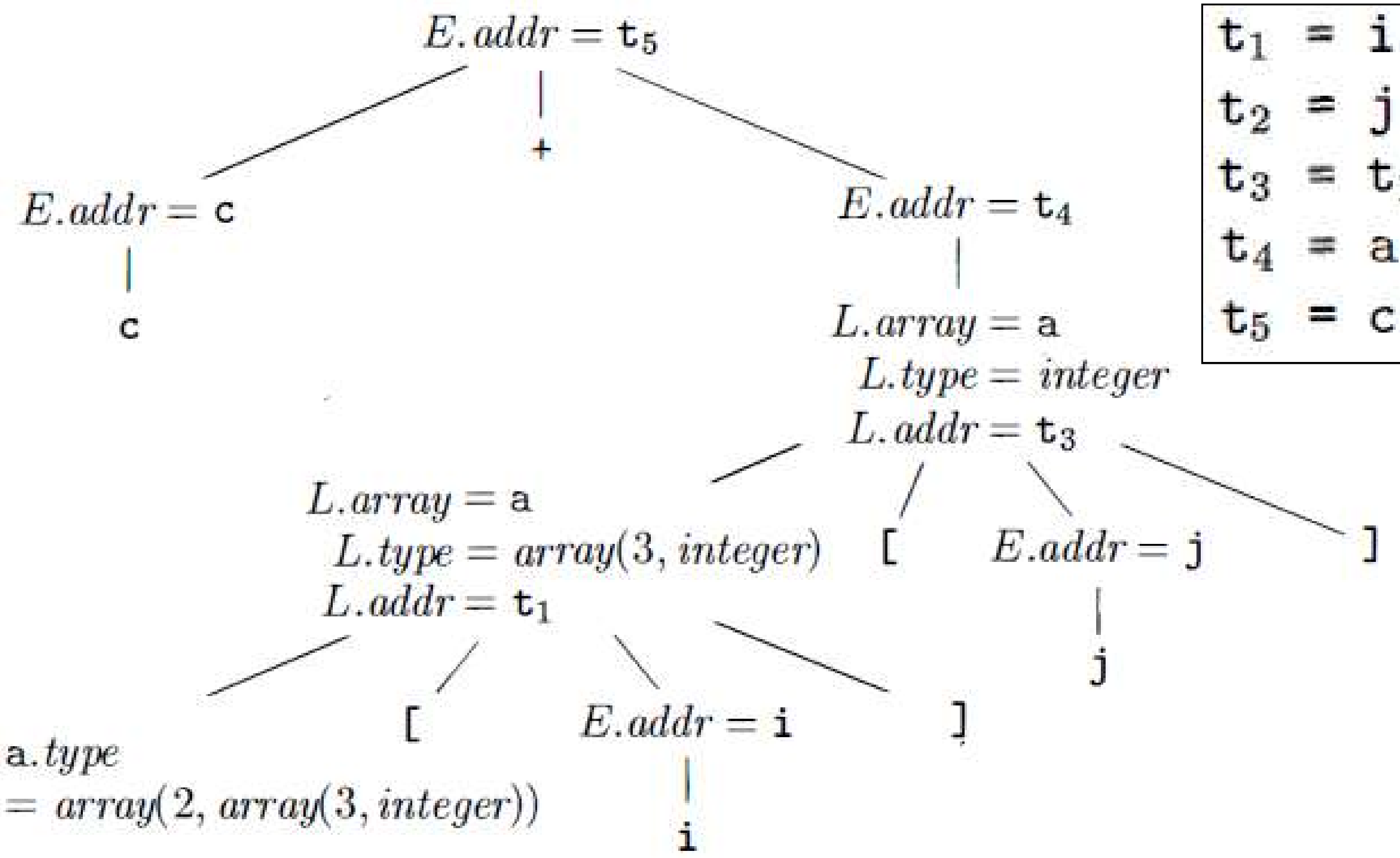
- Nonterminal L has three synthesized attributes:
  1. L.addr denotes a temporary that is used while computing the offset for the array reference by summing the $\mathbf{i_j} \times \mathbf{w_j}$ in (6.4)
  2. L.array is a pointer to the symbol-table entry for the array name.
     - L.array.base is the base address of the array.
     - L.array.type is the type of the array.
  3. L.type is the type of the subarray generated by L.
- Assume t is a type, then
  - t.width represents the width.
  - t.elem gives the element type.

# Example 6.12

**a** is a 2×3 array of integers
**i**, **j**, and **c** are integers

**Annotated parse tree for c + a[i][j]**

**Three-address code for c + a[i][j]**



```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

# 5. Type Checking

- To do type checking a compiler needs to assign a type expression to each component of the source program.
- The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language
- Type checking can take on two forms:
  - Synthesis
  - Inference

# Rules for Type Checking

- Type synthesis builds up the type of an expression from the types of its subexpressions.
- It requires names to be declared before they are used.

**if** f has type s → t **and** x has type s,
**then** expression f (x) has type t

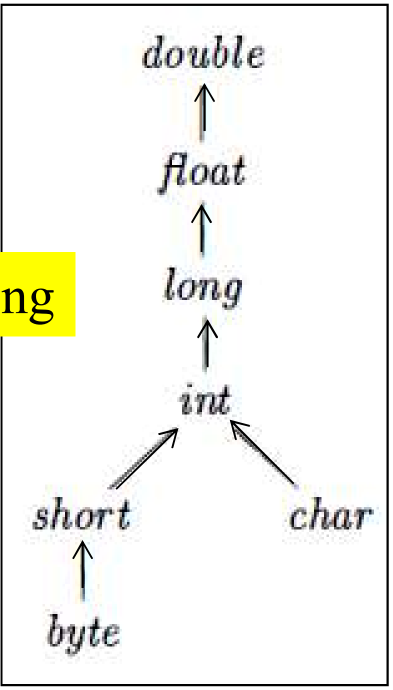
- Type inference determines the type of a language construct from the way it is used.
- It does not require names to be declared

**if** f (x) is an expression,
**then** for some α and β, f has type α → β and x has type α

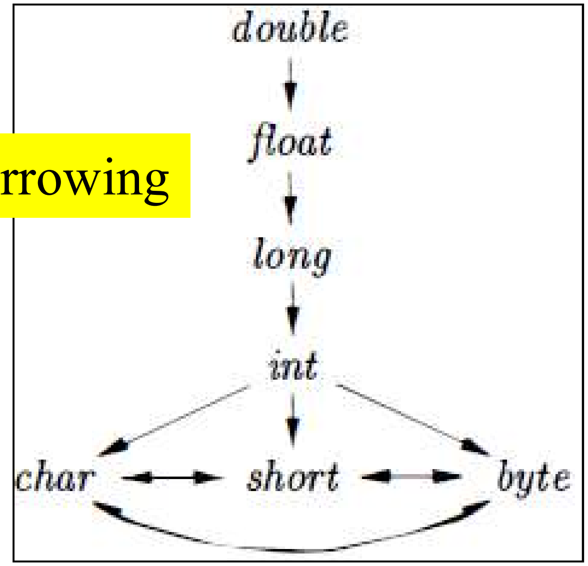
# Type Conversions

- Widening conversions
  - preserve information
- Narrowing conversions
  - lose information
- Coercions (implicit conversions)
  - are done automatically by the compiler.
- Casts (explicit conversions)
  - are done by programmer to write something to cause the conversion.

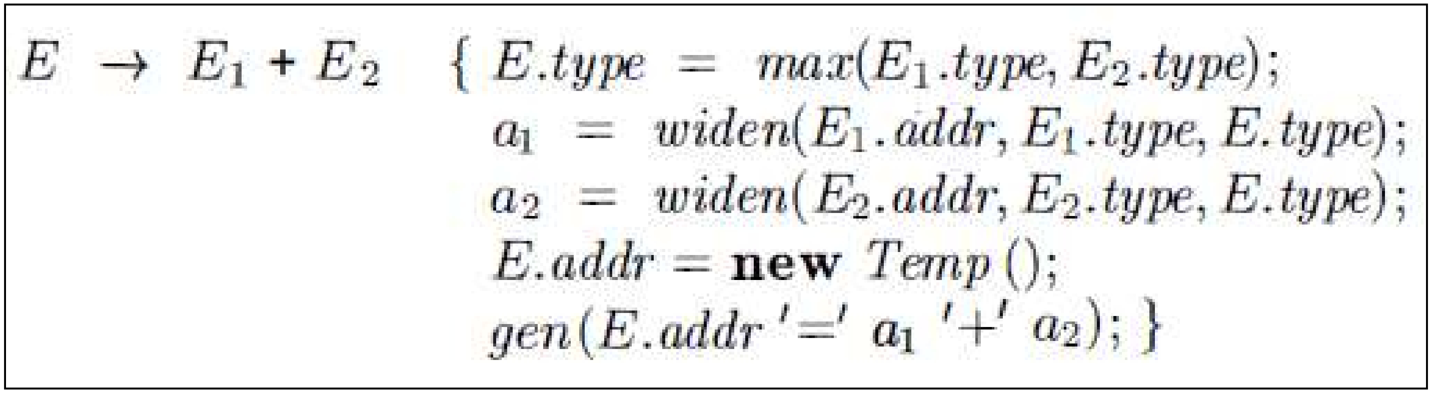
Widening



Narrowing



# Introducing Type Conversions into Expression Evaluation



$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{ \; E.type & = \; max(E_1.type, E_2.type); \\
 & \;\; a_1 & = \; widen(E_1.addr, E_1.type, E.type); \\
 & \;\; a_2 & = \; widen(E_2.addr, E_2.type, E.type); \\
 & \;\; E.addr = \mathbf{new} \; Temp\,(); & \\
 & \;\; gen(E.addr \; '=' \; a_1 \; '+' \; a_2); \; \} &
\end{array}
$$

*max*($t_1$, $t_2$) returns the maximum (or least upper bound) of the two types $t_1$ and $t_2$ in the widening hierarchy.

*widen*(*a*, *t*, *w*) generates type conversions if needed to widen an address *a* of type *t* into a value of type *w*.

x = 2 + 3.14

↓

$t_1$ = (**float**) 2
$t_2$ = $t_1$ + 3.14
x = $t_2$

# Overloading of Functions and Operators

Overloaded function examples

```
void err () { ... }
void err (String s) { … }
```

A type-synthesis rule for overloaded functions

**if** $f$ can have type $s_i \rightarrow t_i$, for $1 \leq i \leq n$, where $s_i \neq s_j$ for $i \neq j$
**and** $x$ has type $s_k$, for some $1 \leq k \leq n$
**then** expression $f(x)$ has type $t_k$

# Type Inference and Polymorphic Functions

The term "polymorphic" refers to any code fragment that can be executed with arguments of different types

ML program for the length of a list

```
fun length(x) =
if null(x) then 0 else length(tl(x)) + 1;
```

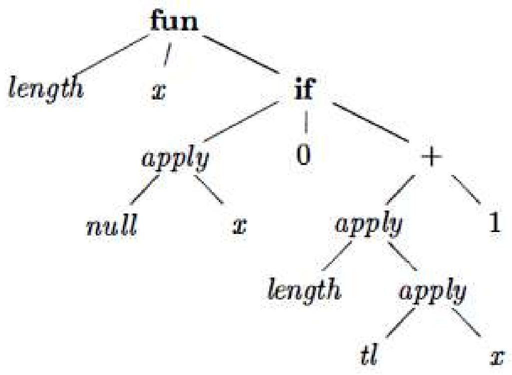
The type of *length*

∀α *list*(α) → *integer*

Example of use of *length*

*length*([“sun”, “mon”, “tue”]) + *length*([10, 9, 8, 7]) returns 7

Abstract syntax tree

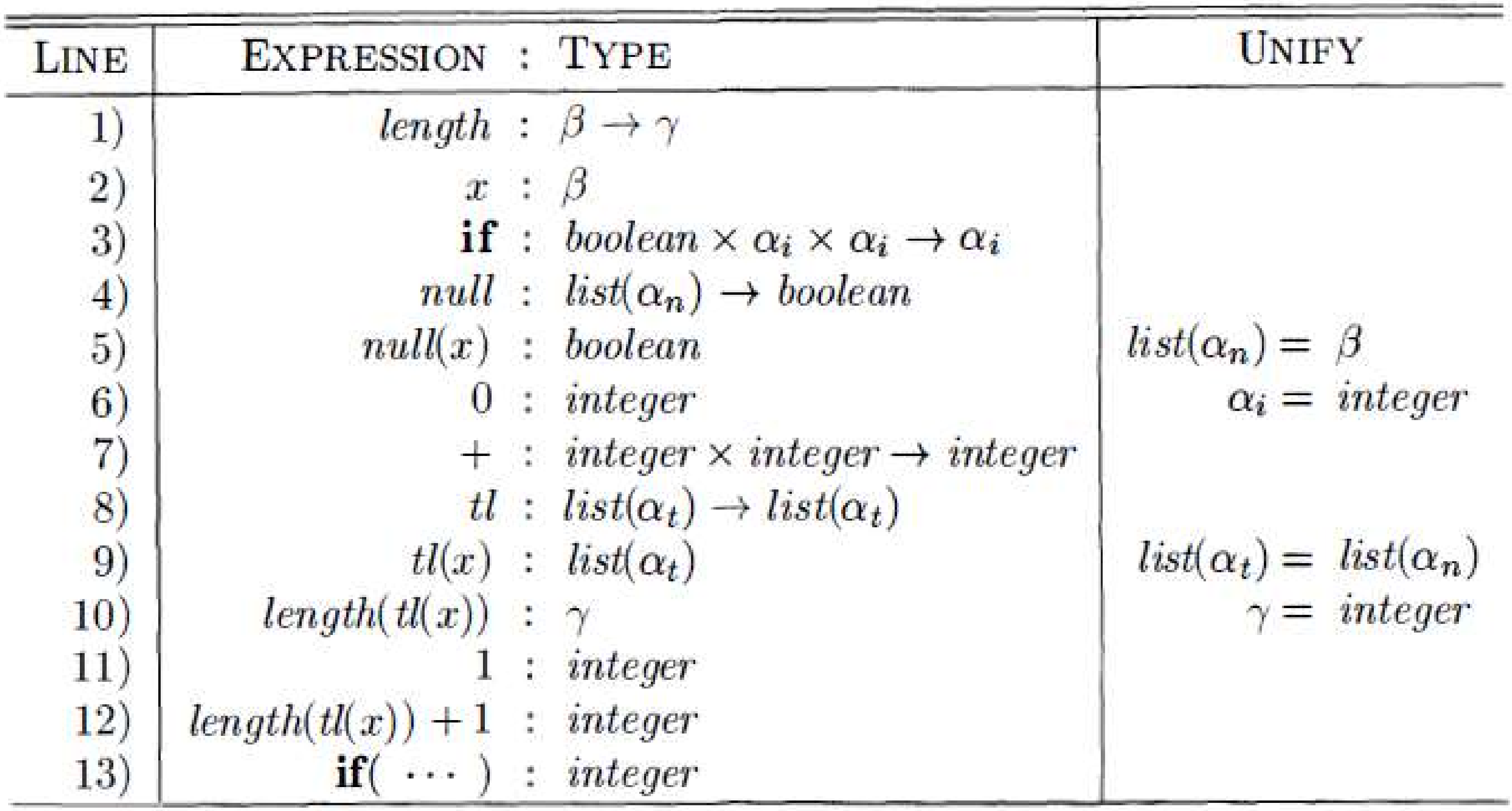


# Substitutions, Instances, and Unification

- A substitution $S$ is a mapping from type variables to type expressions.
  - $S(t)$ = the result of applying the substitution $S$ to the variables in type expression $t$.
    - $S(\alpha) = integer$
    - $t = list(\alpha)$, then $S(t) = list(integer)$
    - $t = \alpha \rightarrow \alpha$, then $S(t) = integer \rightarrow integer$
- $S(t)$ is called an instance of $t$.
- A substitution $S$ is a *unifier* of two types $t_1$ and $t_2$ ($t_1$ and $t_2$ unify), if $S(t_1) = S(t_2)$.
- In the type inference algorithm, we *substitute* type variables by types to create type *instances*

# Inferring a type for the function *length*

**fun** *length*(*x*) = **if** *null*(*x*) then 0 **else** *length*(*tl*(*x*)) + 1;

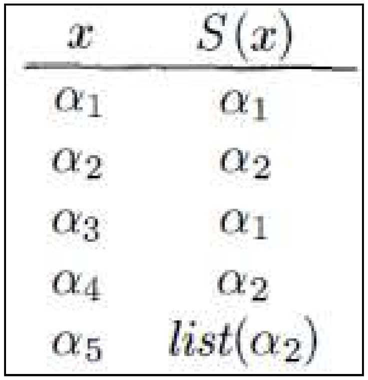


| LINE | EXPRESSION | : | TYPE | UNIFY |
|---|---|---|---|---|
| 1) | $length$ | : | $\beta \rightarrow \gamma$ | |
| 2) | $x$ | : | $\beta$ | |
| 3) | **if** | : | $boolean \times \alpha_i \times \alpha_i \rightarrow \alpha_i$ | |
| 4) | $null$ | : | $list(\alpha_n) \rightarrow boolean$ | |
| 5) | $null(x)$ | : | $boolean$ | $list(\alpha_n) = \beta$ |
| 6) | $0$ | : | $integer$ | $\alpha_i = integer$ |
| 7) | $+$ | : | $integer \times integer \rightarrow integer$ | |
| 8) | $tl$ | : | $list(\alpha_t) \rightarrow list(\alpha_t)$ | |
| 9) | $tl(x)$ | : | $list(\alpha_t)$ | $list(\alpha_t) = list(\alpha_n)$ |
| 10) | $length(tl(x))$ | : | $\gamma$ | $\gamma = integer$ |
| 11) | $1$ | : | $integer$ | |
| 12) | $length(tl(x)) + 1$ | : | $integer$ | |
| 13) | **if**$( \cdots )$ | : | $integer$ | |

⇨ $\forall \alpha_n.\ list(\alpha_n) \rightarrow integer$

# An Algorithm for Unification

Examples 6.18: Consider the two type Expressions $t_1$, $t_2$, and the substitution $S$



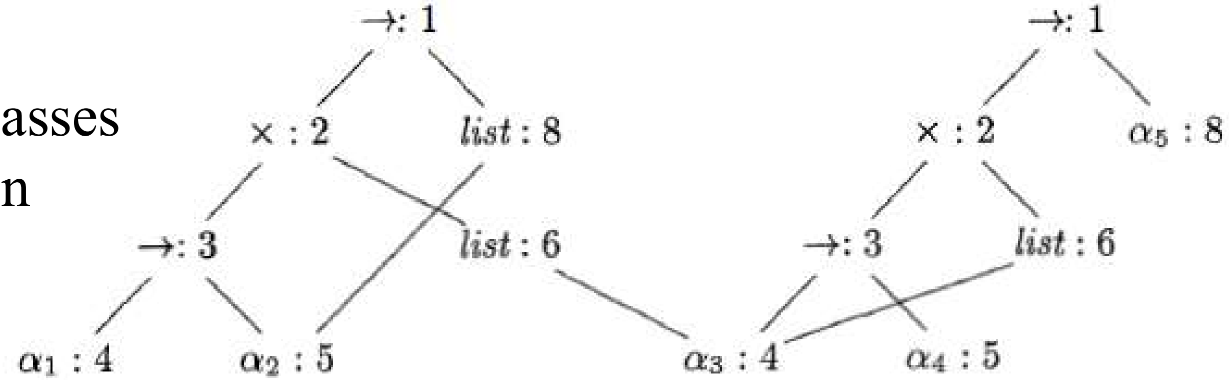
| $x$ | $S(x)$ |
|---|---|
| $\alpha_1$ | $\alpha_1$ |
| $\alpha_2$ | $\alpha_2$ |
| $\alpha_3$ | $\alpha_1$ |
| $\alpha_4$ | $\alpha_2$ |
| $\alpha_5$ | $list(\alpha_2)$ |

$$t_1 = ((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$

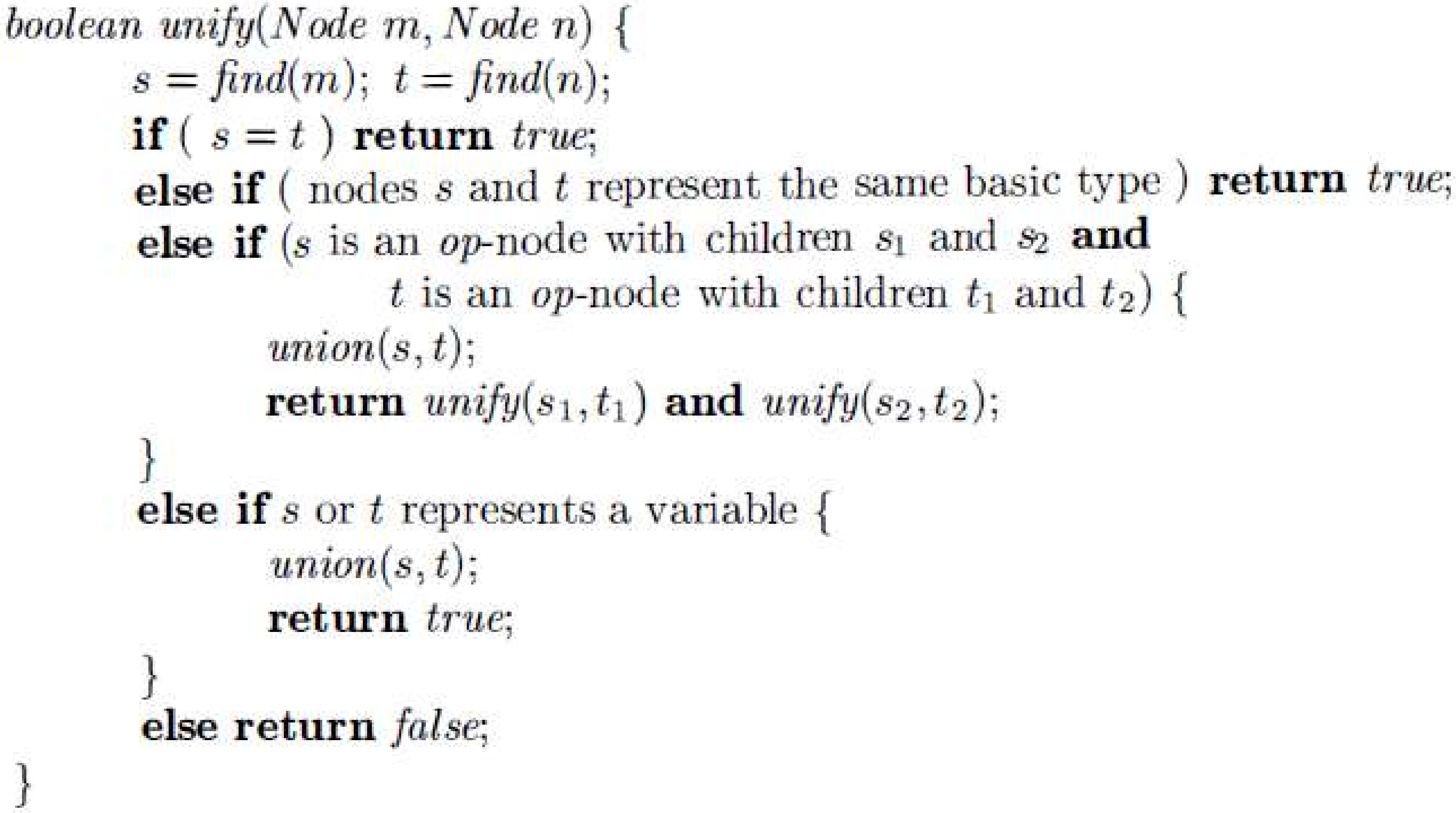
$$t_2 = ((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$$

$$S(t_1) = S(t_2) = ((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_1)) \rightarrow list(\alpha_2)$$

Equivalence classes after unification



# An Algorithm for Unification(Cont.)



```
boolean unify(Node m, Node n) {
        s = find(m);  t = find(n);
        if ( s = t ) return true;
        else if ( nodes s and t represent the same basic type ) return true;
        else if (s is an op-node with children s1 and s2 and
                        t is an op-node with children t1 and t2) {
                union(s, t);
                return unify(s1, t1) and unify(s2, t2);
        }
        else if s or t represents a variable {
                union(s, t);
                return true;
        }
        else return false;
}
```