Semantic Analysis

Compiler Phases and Errors

- Lexical analysis
  - detects inputs with illegal tokens
- Syntax analysis
  - detects inputs with incorrect structure
- Semantic analysis
  - is the last phase of the front end
  - detects all remaining errors

Why Syntax Analysis is not Enough?

- Parsing cannot catch some errors
  - Some language constructs are not context-free
- Example: identifiers are declared before use, i.e.

$$\{wcw | w \in (a|b)^*\}$$

The 1st *w* represents a declaration,
The 2nd *w* represents a use.

Other Semantic Checks?

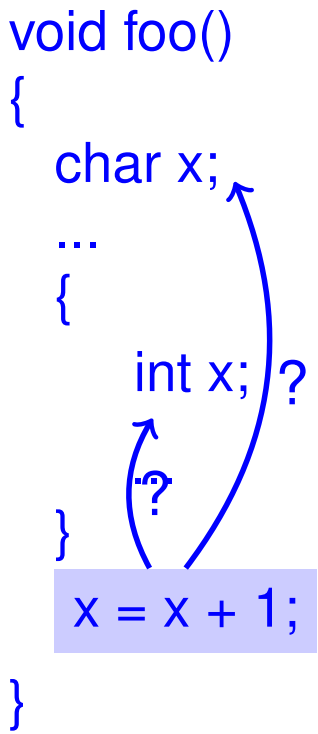
Semantic analyzer also checks

- All identifiers are declared;
- Type consistence;
- Inheritance relationship is correct;
- A class is defined only once;
- A method in a class is defined only once;
- Reserved identifiers are not misused;
- ...

A Simple Semantic Check

"Matching identifier **declarations** with **uses**"

- Important analysis in most languages
- If there are multiple declarations, which one to match?

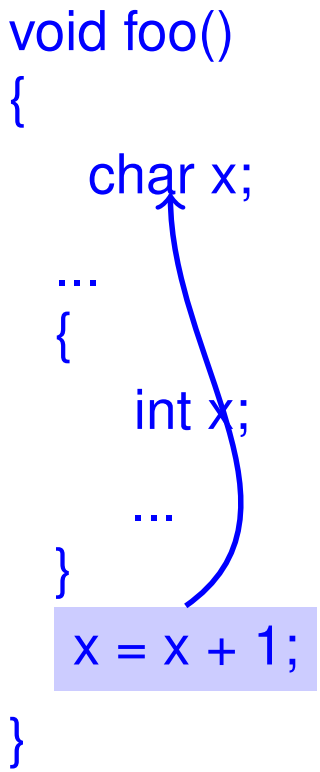


Scope

- The *scope* of an identifier is the portion of a program in which that identifier is accessible
  - The same identifier may refer to different things in different parts of the program
  - Different scopes for *same name* don't overlap
  - An identifier may have restricted scope
- Two types: static scope and dynamic scope

Static Scope

- Static scope depends on the program text, not run-time behavior
  - Most languages have static scope
- Refer to the closest enclosing definition



Dynamic Scope

- A dynamically-scoped variable refers to the closest enclosed binding in the execution of the program
  - LISP, SNOBOL
  - Modern LISP has changed to mostly static scoping

```
void foo()
{
   (1) char x;
   (2) ...
   (3) if (...) {
   (4)     int x;
   (5)       ...
          }
   (6) x = x + 1;
}
```

- Which *x*'s definition is the closest?
  - case (a): ...(1)...(2)...(3)...(6)
  - case (b): ...(1)...(2)...(3)...(4)...(5)...(6)

Symbol Table

Symbol Table

- A symbol table handles creating and merging scopes
- A data structure that tracks information about all identifiers in a program
  - It may not be considered as part of the parser as it is used in all phases of compilation
  - It is a database (sort of)
- Symbol tables are usually maintained during compilation, and discarded after generating the binary code
  - To keep the symbol table for debugging, use “gcc -g ...”

Maintaining Symbol Table

- Basic idea

  int x; ... void foo() { int x; ... x=x+1; } ... x=x+1 ...

  - Before processing *foo*, add definition of *x*, overriding any other definition of *x*
  - After processing *foo*, remove definition of *x* and restore old *x* if any

- Operations

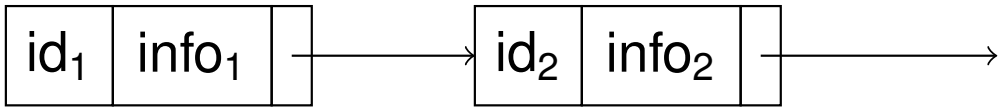
enter_scope()	start a new nested scope
exit_scope()	exit current scope
find_symbol(x)	find the information about *x*
add_symbol(x)	add a symbol *x* to the symbol table
check_symbol(x)	true if x is defined in current scope

Symbol Table Structure

- What data structure to choose ?
  - List
  - Binary tree
  - Hash table
- Tradeoffs: time vs space
  - Let us first consider the organization w/o scope

List

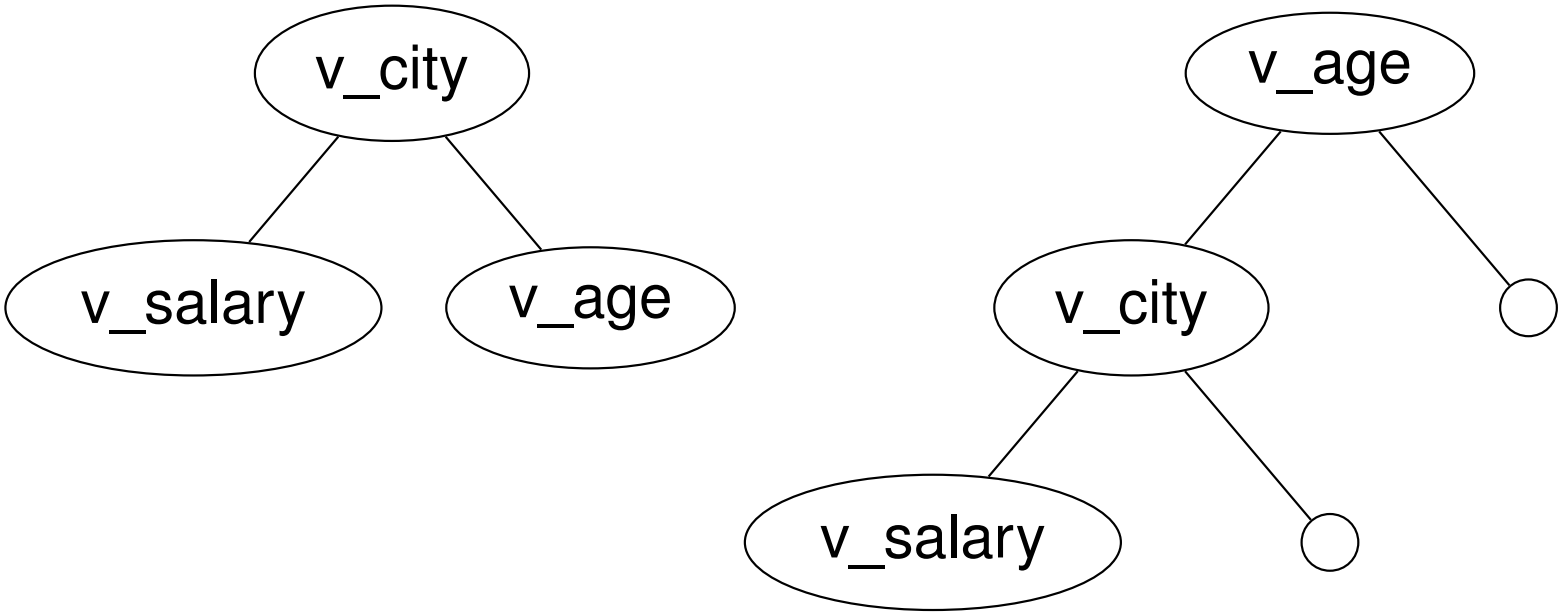
- Linked list or array



id_1	$info_1$
id_2	$info_2$
...	...

- Array: simple, no space wasted, expensive insertion/deletion, search length O(n)
- Linked list: simple, extra pointer space, possible to insert/delete, search length O(n)
  - An optimization: always move the found identifier to the head of the list
  - Frequent used identifiers are found fast

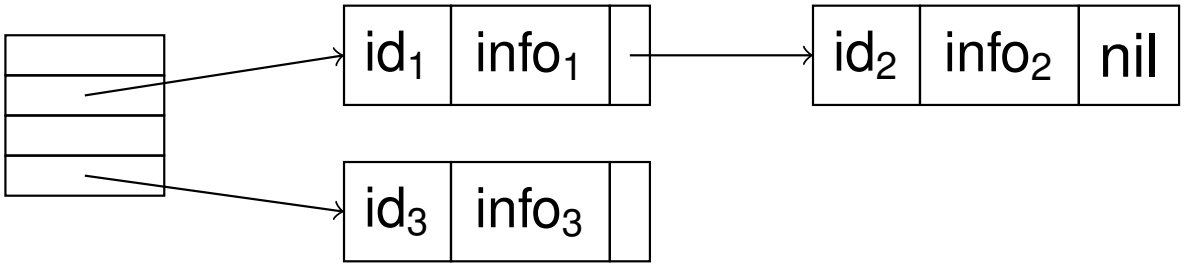
Binary Tree



Discussion:

- Searching on balanced tree proceeds faster
- Use more space than array/list
- In the worst case, tree may reduce to linked list (w/ dummy child pointers)

Hash Table



- $hash(id\_name) \rightarrow index$
  - entries are added when declarations are processed
  - lookup when statements are processed
- $N << M$
  - M: the size of hash table
  - N: the number of stored identifiers

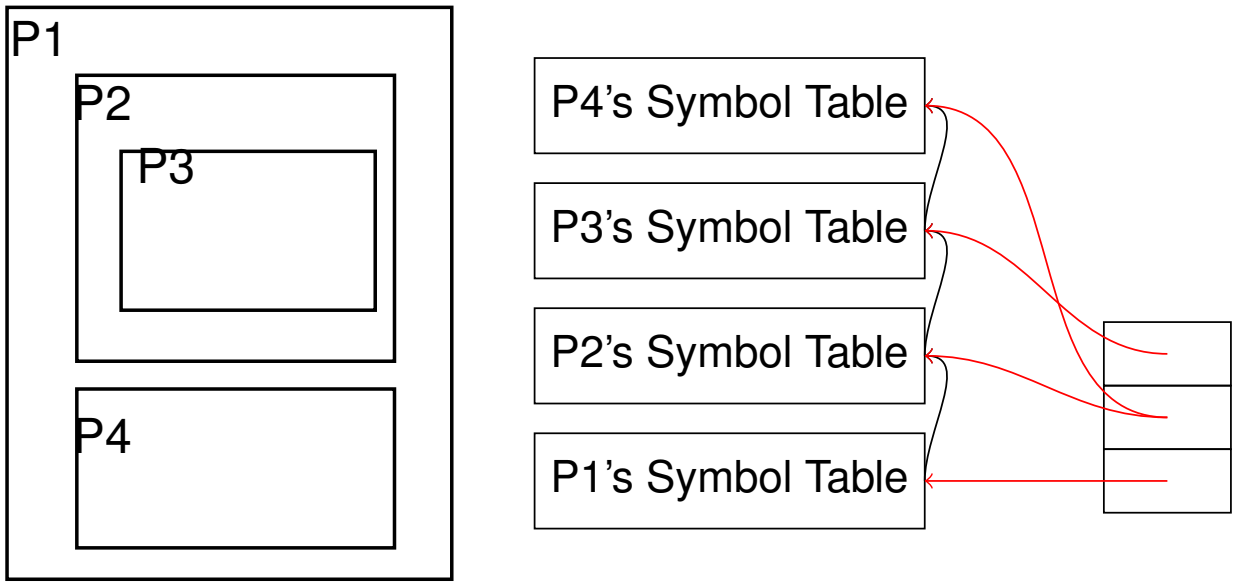
Adding Scope Information to the Symbol Table

- To handle multiple scopes in a program,
  - An individual table for each procedure (or for each scope)
  - Information are added into the table, may not be deleted during compilation

```
class X { ... void f1() {...} ... }
class Y { ... void f2() {...} ... }
X v;
call v.f1();
```

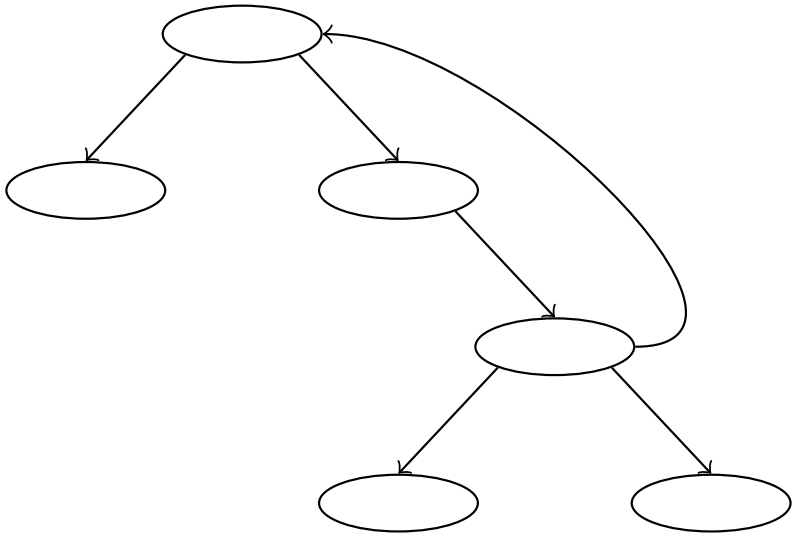
  - Without deleting information, how to enforce scope rule?
    - ☞ Keep a list of all active scopes

Symbol Table with Multiple Scopes



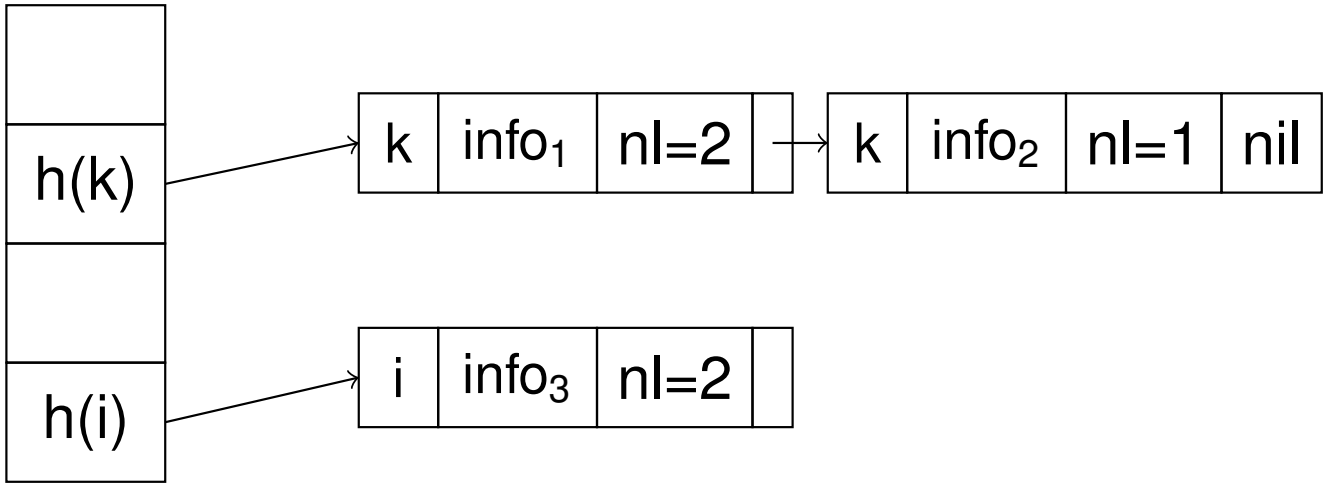
- For the nested function desclaration,
  - Search from top of the active symbol table stack
  - Remove the pointer to the symbol table when exiting its scope

Organizing Multiple Symbol Tables using Binary Tree



- Search in current tree for the given identifier
  - If not found, following the link to find its parent table

Organizing Multiple Symbol Tables using Hash Table



- Link together different entries for the same identifier and associate nesting level with each occurrence of the same name
  - the first one is the latest occurrence of the name, i.e. highest nesting level
  - when exiting level *k*, remove all symbols with level k
  - difficult for dot access (Class.Func)

Check Class Names

- Class names can be used before its definition

```
class c1;
...
c1 v1;
v1.func
```

  - ➢ we cannot check the class in one pass
- Solution
  - ➢ Pass 1: gather all class information
  - ➢ Pass 2: perform the check
- Semantic analysis requires multiple passes

What Information to Store in Symbol Table

- Entry in symbol Table:

string	kind	attributes

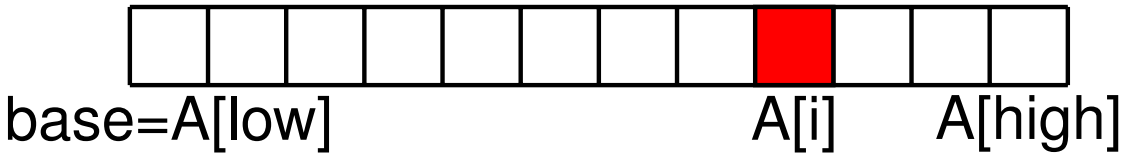
  - String — the name of identifier
  - Kind — label, type name, variable, parameter
- Attributes vary with the kind of symbols
  - label $\rightarrow$ location
  - variable $\rightarrow$ type, address
- Vary with the language
  - Fortran's array $\rightarrow$ type, dimension, dimension size
    real A(3,5) /* required for static allocation */
  - C's array $\rightarrow$ type, dimension, optional dimension size

    ☞ allow both static allocation and dynamic allocation

Addressing Array Elements

int A[low..high];

A[i] ++;



- width — width of each element
- base — address of the first
- low/high — lower/upper bound of subscript

- Addressing an array element:

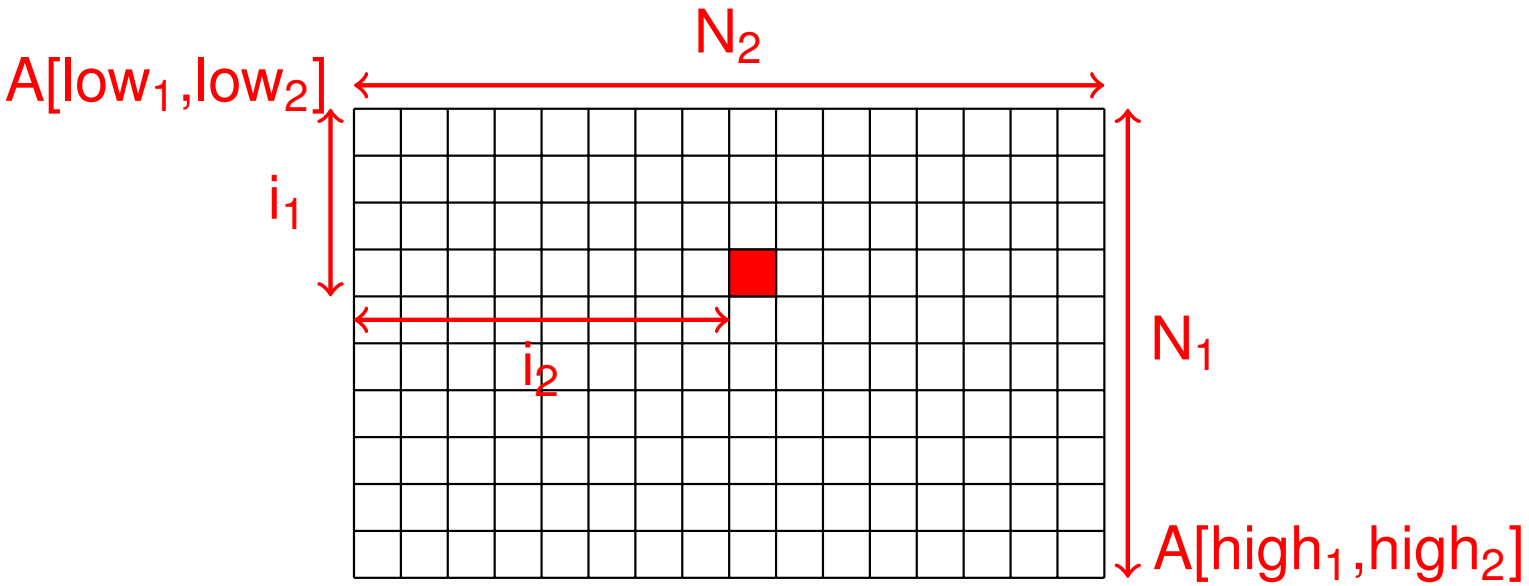
address(A[i])
= base + (i-low) * width = i * width + (base-low*width)
= i * width + C_1

Multi-dimensional Arrays

- Layout n-dimension items in 1-dimension memory

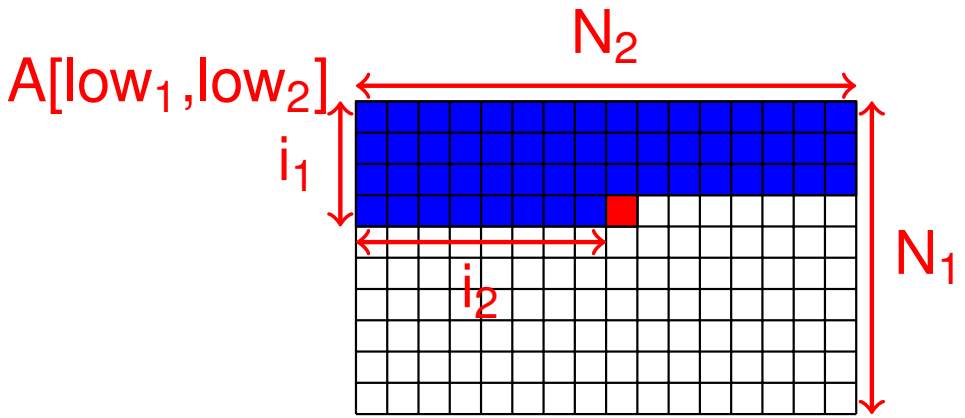
  int A[N_1][N_2]; /* int A[low_1..$high_1$][low_2..$high_2$]; */

  A[i_1][i_2] ++;



Row Major

Row major — store row by row

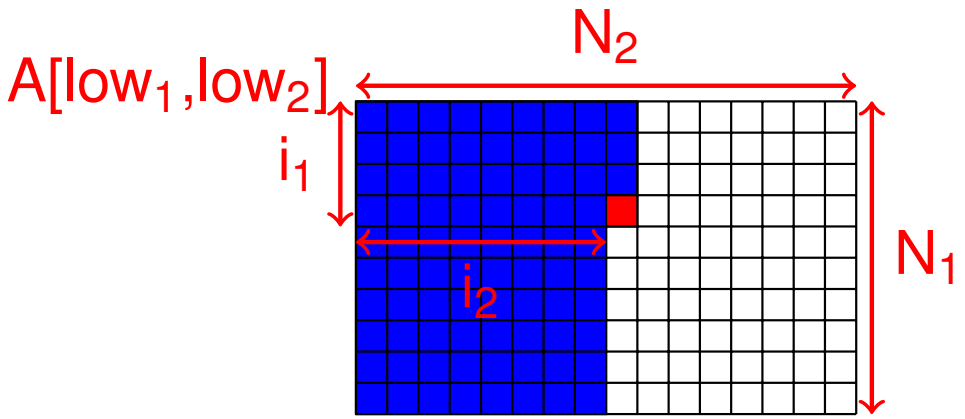


- Need to store "blue" items before $A[i_1,i_2]$

address($A[i_1,i_2]$)
= base + ((i_1-low$_1$) * N_2 + (i_2-low$_2$))* width
= (i_1*N_2 + i_2) * width + C_{2r}

Column Major

Column major — store column by column



- Need to store "blue" items before $A[i_1,i_2]$

address($A[i_1,i_2]$)
= base + ((i_2-low_2) * N_1 + (i_1-low_1))*width
= (i_2*N_1 + i_1) * width + C_{2c}

Generalized Row/Column Major

- Row major: addressing a k-dimension array item
  (low_i = base = 0)
  1-dimension: $A_1 = i_1 * width$
  2-dimension: $A_2 = (i_1 * N_2 + i_2) * width = A_1 * N_2 + i_2 * width$
  3-dimension: $A_3 = A_2 * N_3 + i_3 * width$
  ...
  k-dimension: $A_k = A_{k-1} * N_k + i_k * width$

- Column major: addressing a k-dimension array item
  (low_i = base = 0)
  1-dimension: $A_1 = i_1 * width$
  2-dimension: $A_2 = (i_2 * N_1 + i_1) * width$
  3-dimension: $A_3 = ((i_3 * N_2 + i_2) * N_1 + i_1) * width = i_3 * N_2 * N_1 * width + A_2$
  ...
  k-dimension: $A_k = i_k * N_{k-1} * N_{k-2} * ... * N_1 * width + A_{k-1}$

C's implementation

- C uses row major

```
int fun1(int p[ ][100])
{
...
   int a[100][100];
   a[i1][i2] = p[i1][i2] + 1;
}
```

Why p[][100] is allowed?

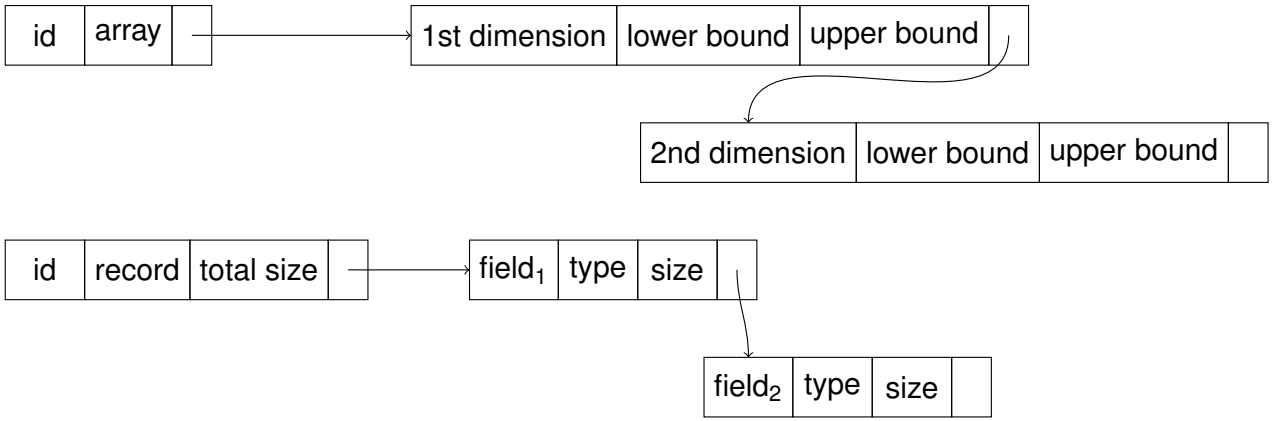
- The information is enough to compute pp[$_1$][$_2$]'s address
- ... $A_2 = (i_1 * N_2 + i_2) * \text{width}$ ...

Why a[][100] is not allowed?

- Need to allocate space, we will discuss this later

More about Information in Symbol Table

- Type information might be complicated
  - In PASCAL: a: array [1..20] of Integer;
    b: array [Mon .. Fri] of Integer;
  - In C:
```
struct {
    int a[10];
    char b;
    real c;
}
```
- Store all useful information in the symbol table



How to use the rich (type) information in Symbol Table ?

Syntax Directed Translation

What is Syntax Directed Translation?

- To drive semantic analysis tasks based on the language's syntax structures

- What semantic tasks?
  - Generate AST (abstract syntax tree)
  - Check type errors
  - Generate intermediate representation (IR)
- What synatx structures?
  - Context free grammar (CFG)
  - Parse tree generated from parser

How to Perform Syntax Directed Translation?

- How?
  - Attach **attributes** to grammar symbols/parse tree
  - Evaluate attribute values using **semantic actions**

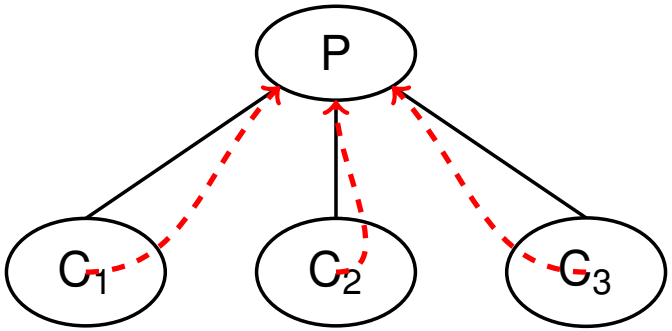
- In project 2, attach attributes to many grammar symbols
  - Representing information about the grammar symbol
    - **tptr** — "tree pointer" of a non-terminal symbol
      ... if **program.tptr** is evaluated, then the parse tree is built
  - Evaluating attributes using semantic rules (actions)
    - { ... $$=makeTree(ProgramOp, leftChild, rightChild); ... }

Attributes?

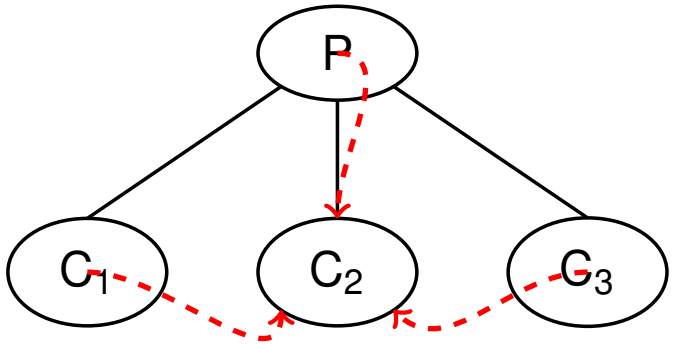
- Attributes can represent anything we need
  - A string
  - A type
  - A number
  - A memory location

Two Types of Attributes

- **Synthesized attributes:** attribute values are computed from some attribute values of its children nodes
  - P.synthesized_attr = f(C_1.attr, C_2.attr, C_3.attr)
- **Inherited attributes:** attribute values are computed from attributes of the siblings and parent of the node
  - C_3.inherited_attr = f(P_1.attr, C_1.attr, C_3.attr)



Synthesized attribute



Inherited attribute

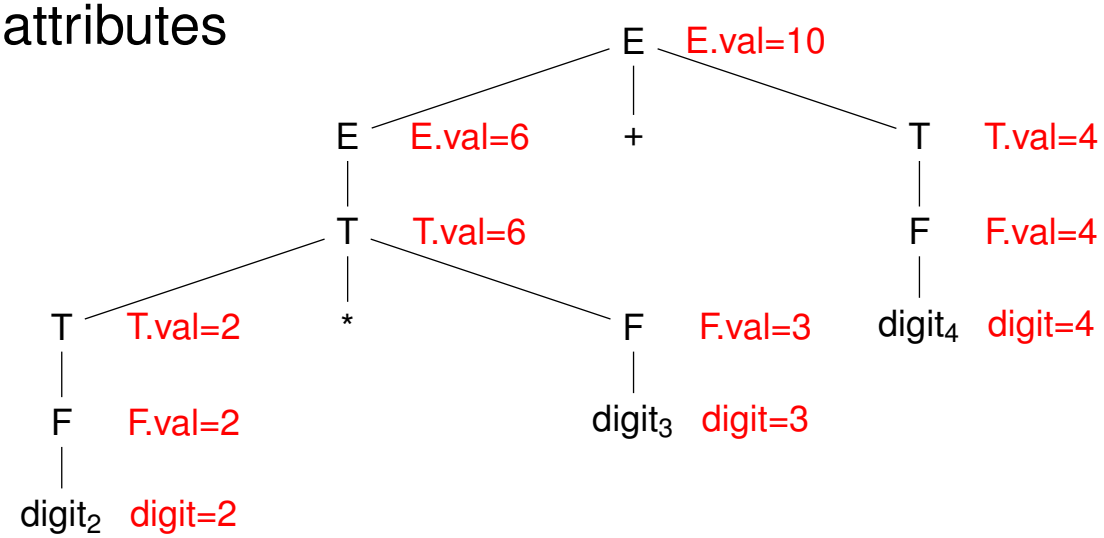
Synthesized Attribute Example

- Example
  - Each non-terminal symbol is associated with **val** attribute
  - Each grammar rule is associated with a **semantic action**

$L \rightarrow E$	{ print(E.val) }
$E \rightarrow E1+T$	{ E.val = E1.val + T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow T1+F$	{ T.val = T1.val + F.val }
$T \rightarrow F$	{ T.val = F.val }
$F \rightarrow (E)$	{ F.val = E.val}
$F \rightarrow digit$	{ F.val = digit.lexval}

Attributed Parse Tree

- Parse tree showing values of attributes
  - annotating or decorating a parse tree with attributes
  - computing attributes at each node
- Properties of attribute parse tree:
  - Terminal symbols — have synthesized attributes only which are usually provided by the lexical analyzer
  - Start symbol — is assumed not to have any inherited attributes



Inherited Attribute Example

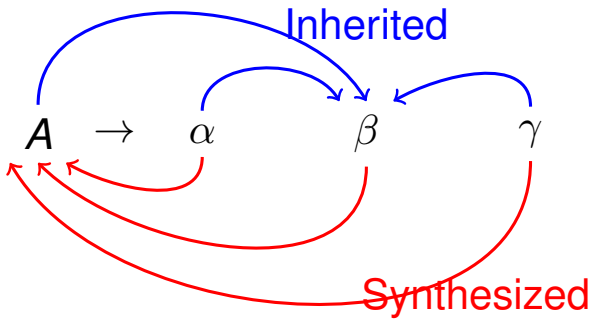
- Example:
  - T — synthesized attribute “type”
  - L — has inherited attribute “in”

$D \rightarrow T\ L$	{ L.in = T.type }
$T \rightarrow$ int	{ T.type = integer }
$T \rightarrow$ real	{ T.type = real }
$L \rightarrow L_1$, id	{ L_1.in = L.in, addtype (id.entry, L.in) }
$L \rightarrow$ id	{ addtype(id.entry, L.in) }

- We can use *inherited attributes* to track **type** information
- We can use *inherited attributes* to track whether an identifier appear on the left or right side of an assignment operator “:=” (e.g. a := a +1)

Attributed Grammar

- Attributed grammar



has semantic rule of the form

$b = f(c_1, c_2, ..., c_n)$

either

1. b is synthesized attributed of A and c_1 ($1 \leq i \leq n$) are attributes of grammar symbols of its Right Hand Side (RHS); or
2. b is an inherited attribute of one of the grammar symbols of RHS of production and c_i's are attribute of A and/or other symbols on the RHS

Two Types of Syntax Directed Translation

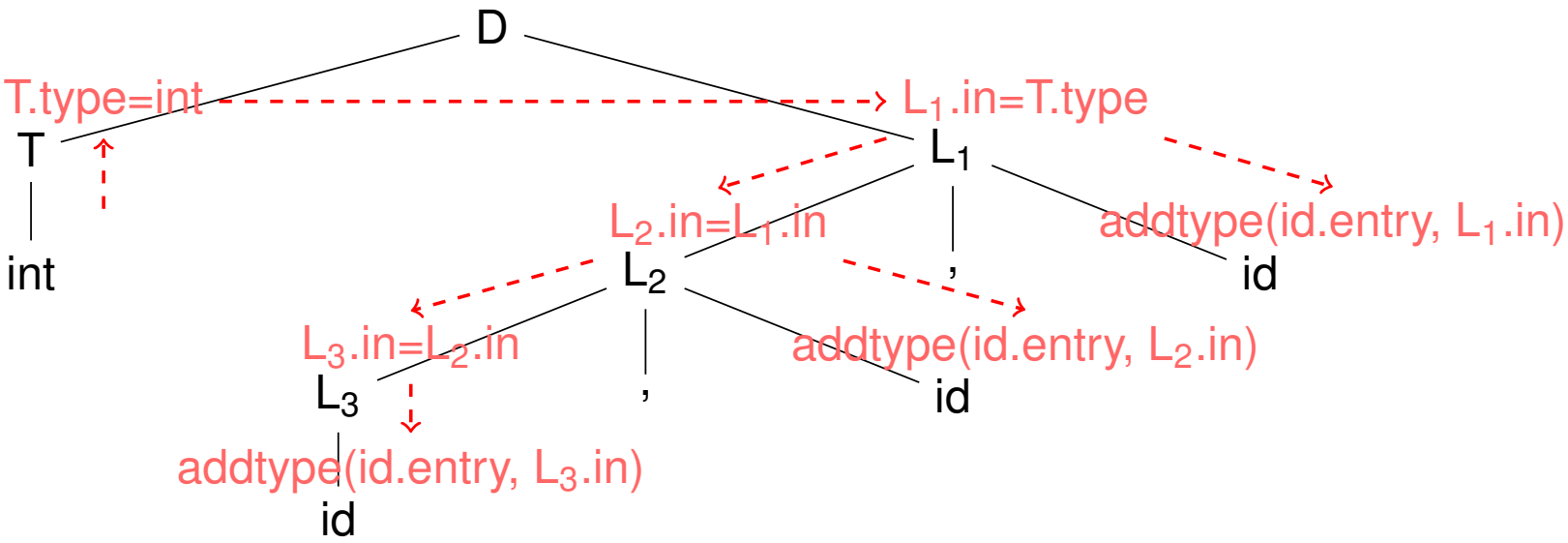
- Syntax directed definitions
  - define high level specifications for translation
  - don't specify order of evaluation/translation
    - hides implementation details
- Syntax directed translation schemes
  - specify the order in which semantic rules are to be evaluated
  - shows more implementation details

Why Evaluation Order is Important?

- Why order is a problem?
  - recall attributed parse tree ...
- Conceptual view of evaluation
  - When parse tree is ready, traverse the tree as needed to evaluate the semantic rules to
    - generate AST node
    - modify symbol table
    - issue error messages as a result of type checking
    - compute information needed by other rules

Dependency Graph

- Based on dependency, we can construct dependency graph which shows the order of evaluation
  - We assume acyclic graph such that there exists a topological order to evaluate attributes
    - i.e. all necessary information is ready when evaluate an attribute at a node



Left-attributed Grammar

- A L-attributed grammar
  - is a class of grammars;
  - may have synthesized attributes;
  - may have inherited attributes that can used evaluated
    - using depth first order of the parse tree
    - from left to right
- Evaluation order is
  - Go down the tree to evaluate the inherited attributes
  - Go up the tree to evaluate the synthesized attributes

Formally,

- A syntax directed translation is L-attributed if each of its attributes is

  either

  - a synthesized attribute of A in $A \rightarrow X_1 ... X_n$,

  or

  - an inherited attribute of X_j in $A \rightarrow X_1 ... X_n$ that
    - depends on attributes of symbols to its left i.e. $X_1 ... X_{j-1}$
    - and/or depends on inherited attributes of A

More Efficient Evaluation?

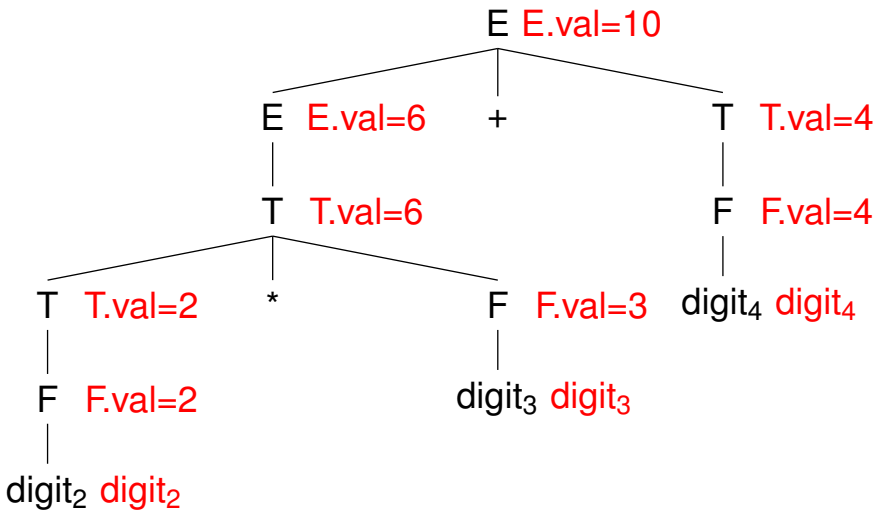
- We discuss how to evaluate after the tree is built
  - find a topological order

- Is that possible to evaluate attributes during parsing?
  - What is the problem to evaluate during parsing
    - The parse tree is not built
    - Some dependency paths may not exist
  - Make sure attributes are available
  - Different parsing schemes see attributes available in different order
    - Top-down parsing — LL(k) parsing
    - Bottum-up parsing — LR(k) parsing

Syntax Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

- it is natural and easy to evaluate synthesized attributes



parsing stack:

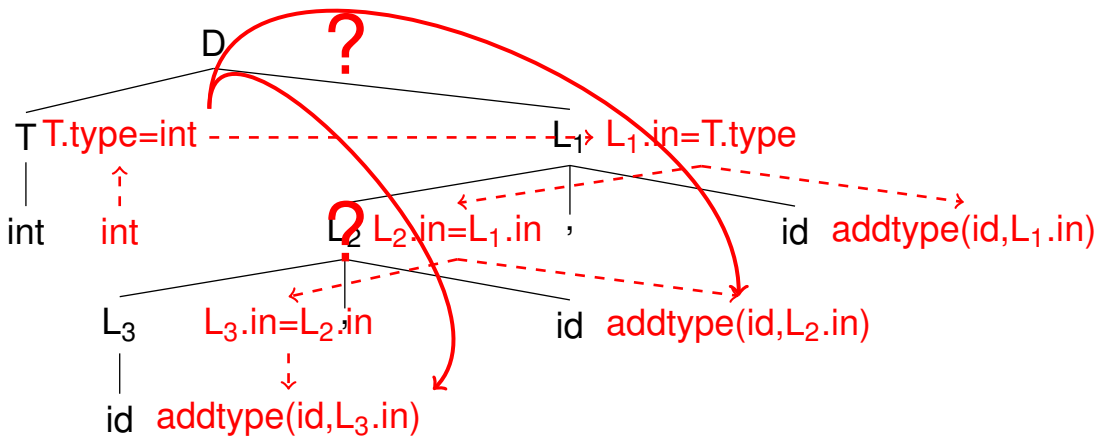
$S_?$	F	F.val=4
$S_?$	+	-
$S_?$	E	E.val=6
$S_?$	$	-

(state) (symbol) (attribute)

Syntax Translation Scheme for Bottom-Up Parsing

When using LR parsing (bottom-up parsing),

- it is **not natural** to evaluate inherited attributes



parsing stack:

$S_?$	id	**id.type=L_3.in**
$S_?$	T	T.type=int
$S_?$	$	-

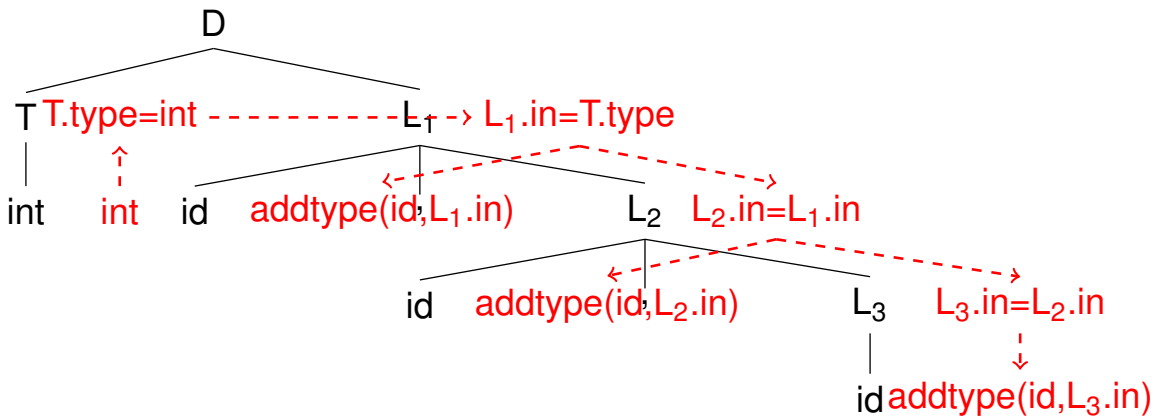
(state) (symbol) (attribute)

- How about Top-Down Parsing?

Syntax Translation Scheme for Top-Down Parsing

When using LL parsing (top-down parsing),

- it is natural to evaluate inherited attributes



parsing stack:

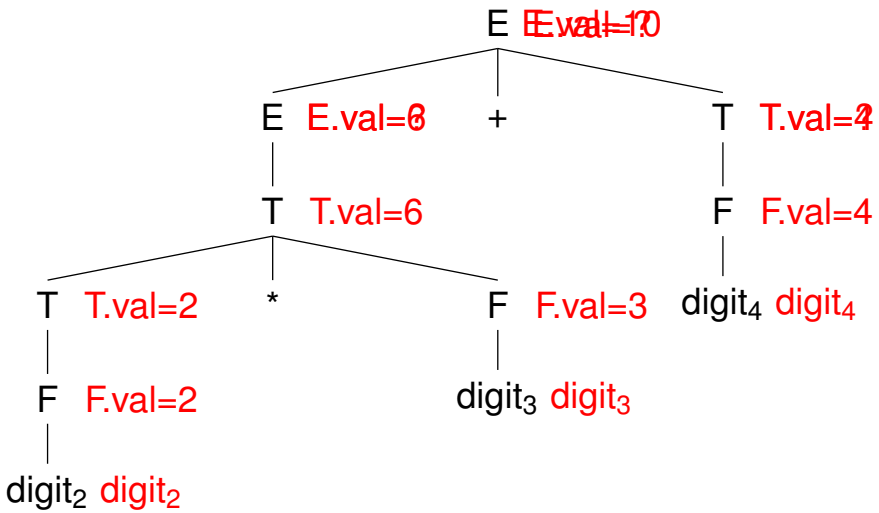
id	id.type=L_1.in
T	T.type=int
D	L_2.in=int

(symbol) (attribute)

Syntax Translation Scheme for Top-Down Parsing

When using LL parsing (top-down parsing),

- it is **not natural** to evaluate synthesized attributes



parsing stack:

T	T.val=?
+	
E	E.val=?

(symbol) (attribute)

- How to
  - Evaluate inherited attributes using bottom-up parsing?
  - Evaluate synthesized attributes using top-down parsing?

Rewriting Attributed Grammar

- First, let us study how to evaluate **synthesized attributes** using **top-down parsing**

- Solution: rewrite the grammar to introduce inherited attributes to assist the evaluation
  - Recall that top-down parser naturally evaluates inherited attributes

L → E	{ print(E.val) }
E → E1+T	{ E.val = E1.val + T.val }
E → T	{ E.val = T.val }
T → T1+F	{ T.val = T1.val + F.val }
T → F	{ T.val = F.val }
F → (E)	{ F.val = E.val}
F → digit	{ F.val = digit.lexval}

Example of Grammar Rewriting

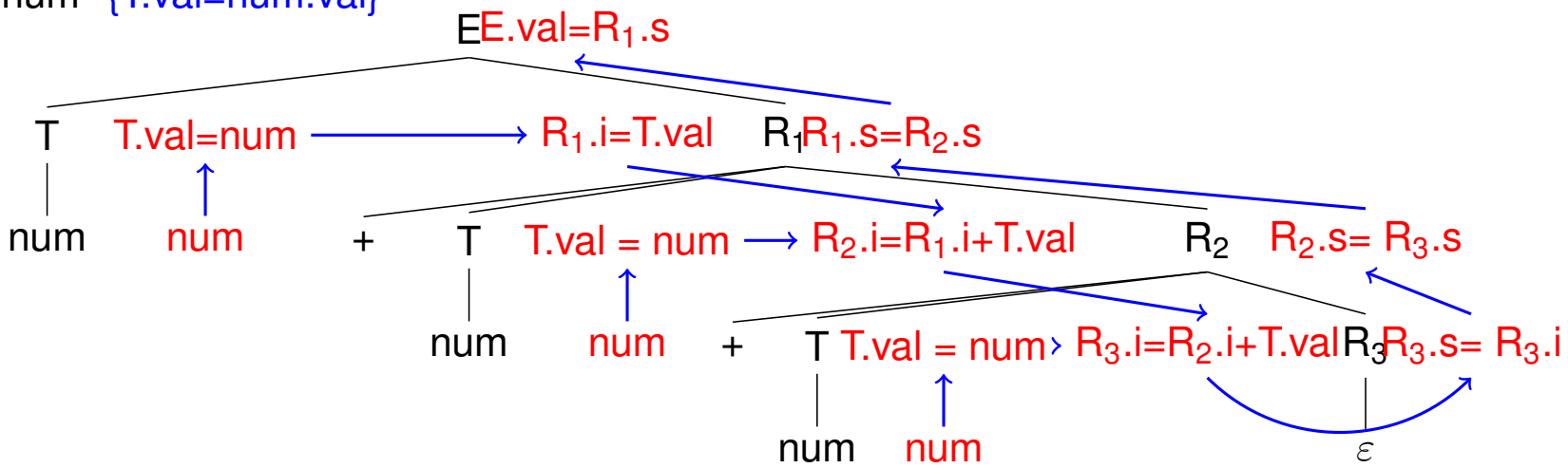
$$
\begin{aligned}
E &\rightarrow T \quad R \\
R &\rightarrow + \quad T \quad R \\
R &\rightarrow - \quad T \quad R \\
R &\rightarrow \varepsilon \\
T &\rightarrow (\; E \;) \\
T &\rightarrow \text{num}
\end{aligned}
$$

- Both inherited and synthesized attributes are used
  - T — synthesized attribute T.val
  - R — inherited attribute R.i
    synthesized attribute R.s
  - E — synthesized attribute E.val

Translation Scheme for the Example

- Evaluating synthesized attribute with the introduction of new inherited attribute

$E \rightarrow T \ \{R.i=T.val\} \ R \ \{E.i=R.s\}$
$R \rightarrow + \ T \ \{R.i=R.i+T.val\} \ R_1 \ \{R.s=R_1.s\}$
$R \rightarrow - \ \{R.i=R.i-T.val\} \ R_1 \ \{R.s=R_1.s\}$
$R \rightarrow \varepsilon \ \{R.s=R_1.s\}$
$T \rightarrow (\ E \ ) \ \{T.val=E.val\}$
$T \rightarrow num \ \{T.val=num.val\}$



Evaluating Inherited Attributes using LR

- Recall
  - LR parser naturally evaluates synthesized attributes
  - It is not natural to evaluate inherited attributes
  - We focus on L-attributed grammars
    - What is L-attributed grammar
- **Claim:** the information is in the stack, we just do not know the exact location
- **Solution:** let us hack the stack to find the location

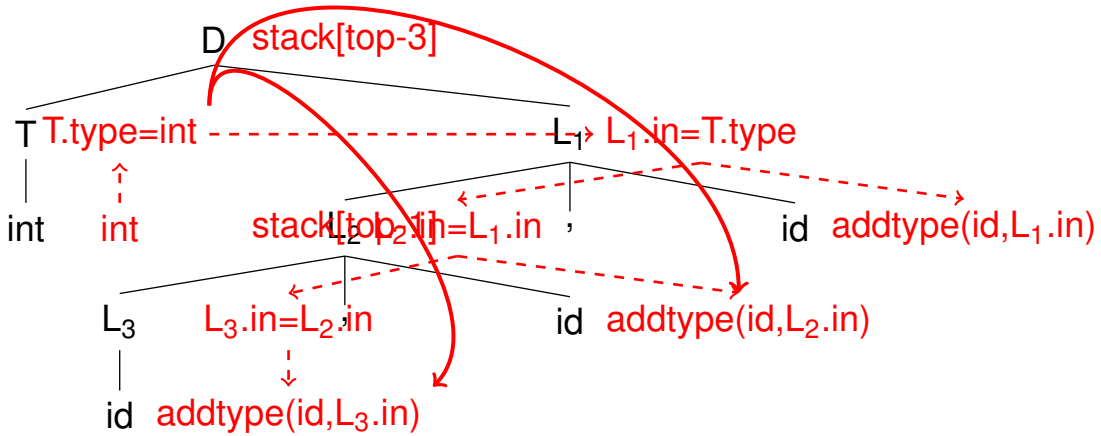
$D \rightarrow T \quad L$

$T \rightarrow \text{int} \quad \{stack[top]=integer\}$

$T \rightarrow \text{real} \quad \{stack[top]=real\}$

$L \rightarrow L \ , \quad \text{id} \quad \{addtype(stack[top],stack[top-3])\}$

$L \rightarrow \text{id} \quad \{addtype(stack[top],stack[top-1])\}$



parsing stack:

$S_?$	id	**id.type=stack[top-3]**
$S_?$	,	
$S_?$	id	**id.type=stack[top-1]**
$S_?$	L_3	L_3.in=int
$S_?$	T	T.type=int
$S_?$	$	-

(state) (symbol) (attribute)

Marker

- Adding "M$\rightarrow \varepsilon$" to assist evaluation

convert

A$\rightarrow$ X {rule} Y

to

A$\rightarrow$ X M Y
M$\rightarrow \varepsilon$ {rule}

- evaluating attributes at the end of RHS, i.e. during reduction
- a marker intuitively *marks* a stack location

Example

- How to add the marker ?

Example 1:

$S \rightarrow a\ A\ \{\ C.i = A.s\ \}\ C$
$S \rightarrow b\ A\ B\ \{\ C.i = A.s\ \}\ C$
$C \rightarrow c\ \{\ C.s = f(C.i)\ \}$

Solution:

$S \rightarrow a\ A\ \{\ C.i = A.s\ \}\ C$
$S \rightarrow b\ A\ B\ \{\ M.i=A.s\ \}\ M\ \{\ C.i = M.s\ \}\ C$
$C \rightarrow c\ \{\ C.s = f(C.i)\ \}$
$M \rightarrow \varepsilon\ \{\ M.s = M.i\ \}$

That is:

$S \rightarrow a\ A\ C$
$S \rightarrow b\ A\ B\ M\ C$
$C \rightarrow c\ \{\ C.s = f(stack[top-1])\ \}$
$M \rightarrow \varepsilon\ \{\ M.s = stack[top-2]\ \}$

How to Add the Marker?

1. Identify the stack location(s) to find the desired attribute
2. Is there a conflict of location?
   - ➢ Yes, add a marker;
   - ➢ No, no need to add.
3. Add the marker in the place to remove location inconsistency

Example:

$S \rightarrow a\ A\ B\ C\ E\ D$
$S \rightarrow b\ A\ F\ B\ C\ F\ D$
$C \rightarrow c$ {/* C.s = f(A.s) */}
$D \rightarrow d$ {/* D.s = f(B.s) */}

Answer

S → a A B C E D
S → b A D M B C F D
C → c {/* C.s = f(stack[top-2]) */}
D → d {/* D.s = f(stack[top-3]) */}
M → ε {/* M.s = f(stack[top-2]) */}

- Regarding C.s, from stack[top-2], and stack[top-3]
  .... add a Marker
- Regarding D.s, always from stack[top-2]
  ... no need to add