




Semantic Analysis

Compiler Phases and Errors

-  Lexical analysis
-  Syntax analysis
-  Semantic analysis

Compiler Phases and Errors

- ❏ Lexical analysis
 - detects inputs with illegal tokens
- ❏ Syntax analysis
- ❏ Semantic analysis

Compiler Phases and Errors

- ❏ Lexical analysis
 - detects inputs with illegal tokens
- ❏ Syntax analysis
 - detects inputs with incorrect structure
- ❏ 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.

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

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?

A Simple Semantic Check

“Matching identifier **declarations** with **uses**”

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

```
void foo()
{
    char x;

    ...
    {
        int x;

        ...
    }
    x = x + 1;
}
```

A Simple Semantic Check

“Matching identifier **declarations** with **uses**”

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

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

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

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

Static Scope

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

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

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;  
}
```

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 ?

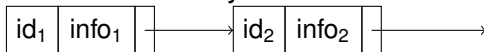
- List
- Binary tree
- Hash table

❏ Tradeoffs: time vs space

- Let us first consider the organization w/o scope

List

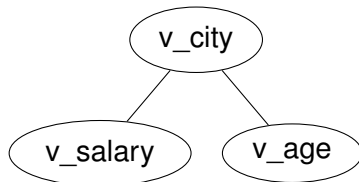
Linked list or array



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

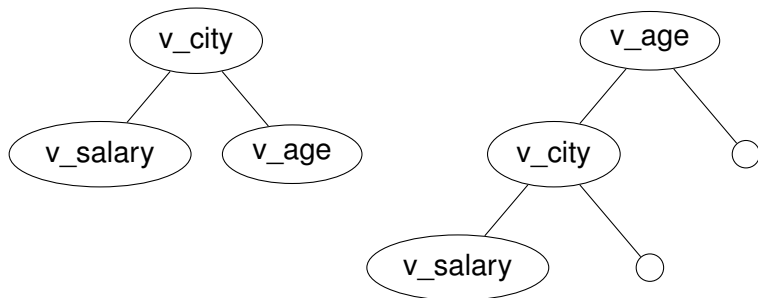
Binary Tree



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)

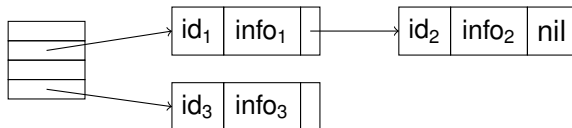
Binary Tree



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

❏ $N \ll M$

- M: the size of hash table
- N: the number of stored identifiers

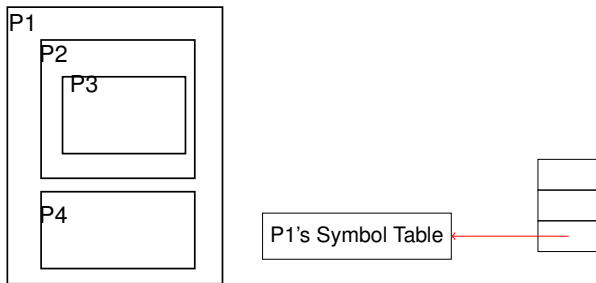
Adding Scope Information to the Symbol Table

- ❏ 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

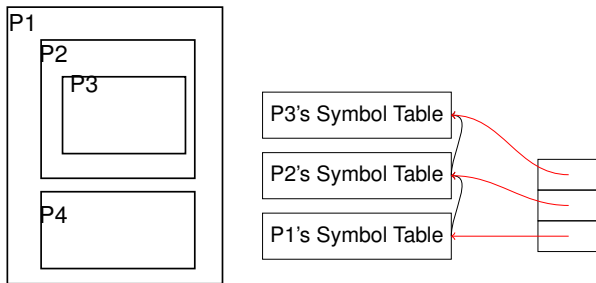
Symbol Table with Multiple Scopes



For the nested function declaration,

- Search from top of the active symbol table stack
- Remove the pointer to the symbol table when exiting its scope

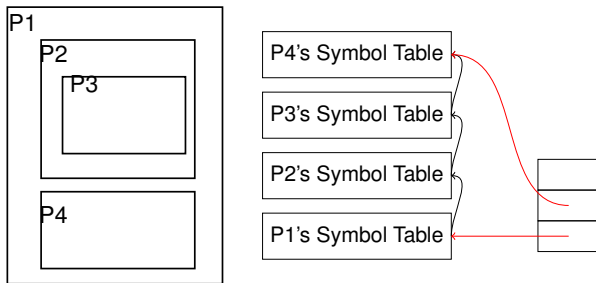
Symbol Table with Multiple Scopes



For the nested function declaration,

- Search from top of the active symbol table stack
- Remove the pointer to the symbol table when exiting its scope

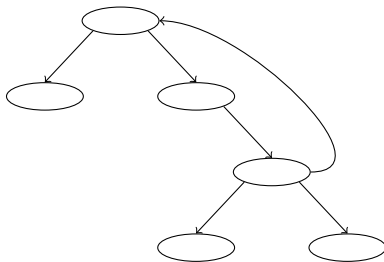
Symbol Table with Multiple Scopes



For the nested function declaration,

- Search from top of the active symbol table stack
- Remove the pointer to the symbol table when exiting its scope

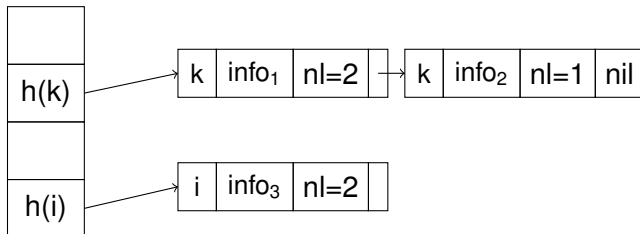
Organizing Multiple Symbol Tables using Binary Tree



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 → location
- variable → type, address

□ Vary with the language

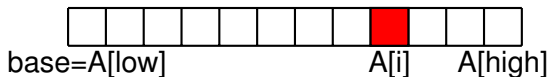
- Fortran's array → type, dimension, dimension size
`real A(3,5) /* required for static allocation */`
- C's array → type, dimension, optional dimension size

☞ 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

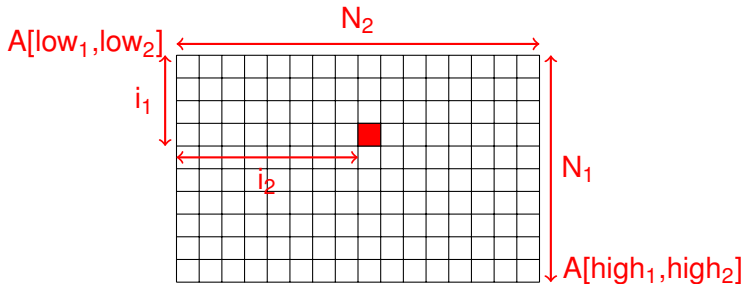
 Addressing an array element:

$$\begin{aligned} &\text{address}(A[i]) \\ &= \text{base} + (i - \text{low}) * \text{width} = i * \text{width} + (\text{base} - \text{low} * \text{width}) \\ &= i * \text{width} + C_1 \end{aligned}$$

Multi-dimensional Arrays

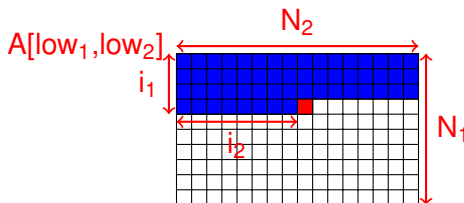
- Layout n-dimension items in 1-dimension memory

```
int A[N1][N2]; /* int A[low1..high1][low2..high2]; */  
A[i1][i2] ++;
```



Row Major

Row major — store row by row

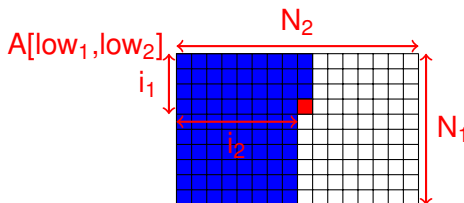


 Need to store “blue” items before $A[i_1, i_2]$

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

Column Major

Column major — store column by column



 Need to store “blue” items before $A[i_1, i_2]$

$$\begin{aligned}
 &\text{address}(A[i_1, i_2]) \\
 &= \text{base} + ((i_2 - \text{low}_2) * N_1 + (i_1 - \text{low}_1)) * \text{width} \\
 &= (i_2 * N_1 + i_1) * \text{width} + C_{2c}
 \end{aligned}$$

Generalized Row/Column Major

□ Row major: addressing a k-dimension array item

(low_i = base = 0)

1-dimension: $A_1 = i_1 * \text{width}$

2-dimension: $A_2 = (i_1 * N_2 + i_2) * \text{width} = A_1 * N_2 + i_2 * \text{width}$

3-dimension: $A_3 = A_2 * N_3 + i_3 * \text{width}$

...

k-dimension: $A_k = A_{k-1} * N_k + i_k * \text{width}$

□ Column major: addressing a k-dimension array item

(low_i = base = 0)

1-dimension: $A_1 = i_1 * \text{width}$

2-dimension: $A_2 = (i_2 * N_1 + i_1) * \text{width}$

3-dimension: $A_3 = ((i_3 * N_2 + i_2) * N_1 + i_1) * \text{width} = i_3 * N_2 * N_1 * \text{width} + A_2$

...

k-dimension: $A_k = i_k * N_{k-1} * N_{k-2} * \dots * N_1 * \text{width} + A_{k-1}$

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?

Why a[][100] is not allowed?

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[i1][i2]`'s address
- ... $A_2 = (i_1 * N_2 + i_2) * \text{width}$...

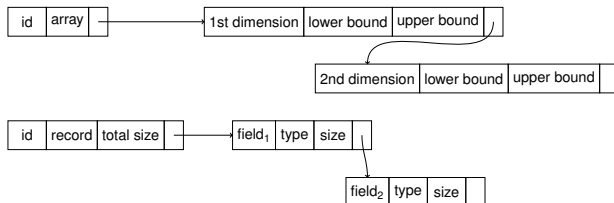
Why `a[][100]` is not allowed?

- 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 syntax 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

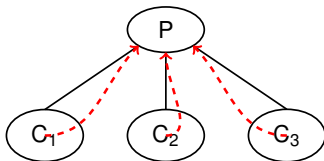
Two Types of Attributes

- ❑ **Synthesized attributes:** attribute values are computed from some attribute values of its children nodes

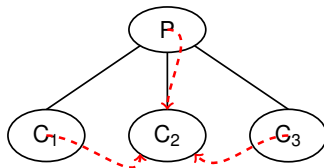
➤ $P.\text{synthesized_attr} = f(C_1.\text{attr}, C_2.\text{attr}, C_3.\text{attr})$

- ❑ **Inherited attributes:** attribute values are computed from attributes of the siblings and parent of the node

➤ $C_3.\text{inherited_attr} = f(P.\text{attr}, C_1.\text{attr}, C_3.\text{attr})$



Synthesized attribute



Inherited attribute

Synthesized Attribute Example

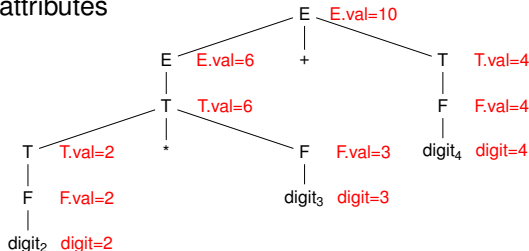
Example

- 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 }
$F \rightarrow \text{digit}$	{ 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:

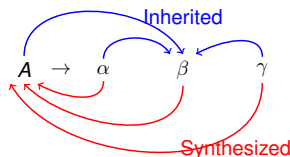
- T — synthesized attribute “type”
- L — has inherited attribute “in”

```
D → T L    { L.in = T.type }  
T → int    { T.type = integer }  
T → real   { T.type = real }  
L → L1 , id { L1.in = L.in, addtype (id.entry, L.in) }  
L → 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

$$b = f(c_1, c_2, \dots, c_n)$$

either

1. b is synthesized attribute of A and c_i ($1 \leq i \leq n$) 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 ...

Why Evaluation Order is Important?

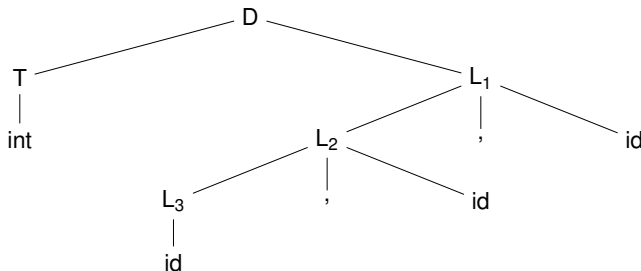
- ❏ Why order is a problem?
 - recall attributed parse tree ...

- ❏ 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

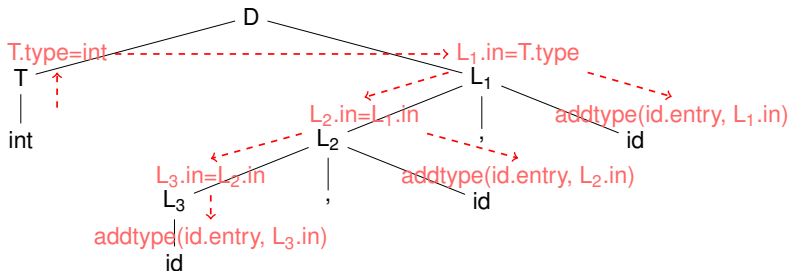
Dependency Graph

- 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



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 $A \rightarrow X_1 \dots X_n$,

or

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

More Efficient Evaluation?

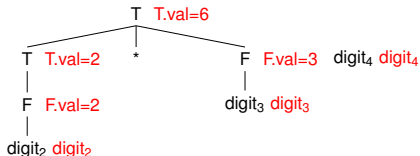
- ❏ We discuss how to evaluate after the tree is built
 - find a topological order

- ❏ 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
 - Bottom-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:

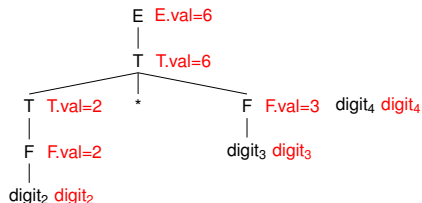
S ₇	T	T.val=6
S ₇	\$	-

(state) (symbol) (attribute)

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:

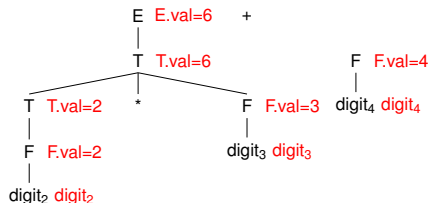
S ₇	E	E.val=6
S ₇	\$	-

(state) (symbol) (attribute)

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:

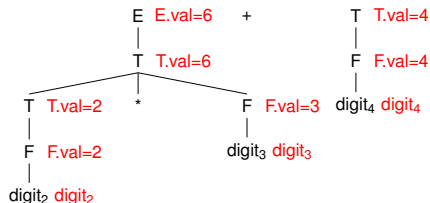
S ₇	F	F.val=4
S ₇	+	-
S ₇	E	E.val=6
S ₇	\$	-

(state) (symbol) (attribute)

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:

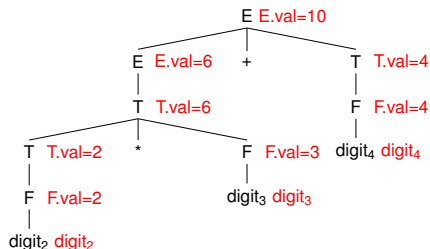
S ₇	T	T.val=4
S ₇	+	-
S ₇	E	E.val=6
S ₇	\$	-

(state) (symbol) (attribute)

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:

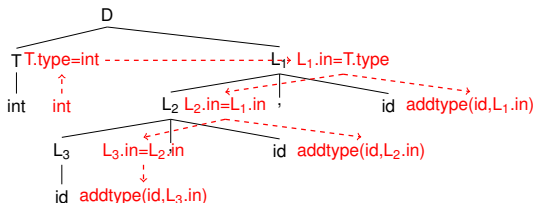
S ₇	E	E.val=10
S ₇	\$	-

(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?	\$	-

(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

int

,

id

,

id

id

parsing stack:

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

T.type=int
|
int ↑
 int

,

id

,

id

id

parsing stack:

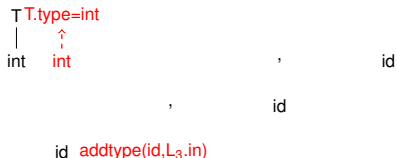
S ₇	T	T.type=int
S ₇	\$	-

(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:

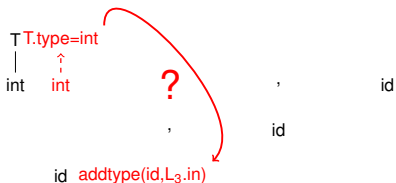
S_7	id	$id.type=L_3.in$
S_7	T	$T.type=int$
S_7	$\$$	-

(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:

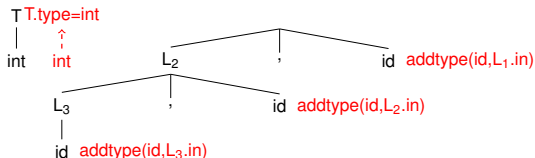
S_7	id	id.type=L ₃ .in
S_7	T	T.type=int
S_7	\$	-

(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:

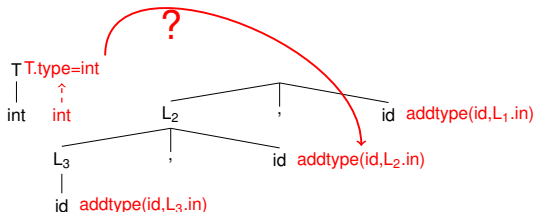
S ₇	id	id.type=L ₃ .in
S ₇	T	T.type=int
S ₇	\$	-

(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 _?	\$	-

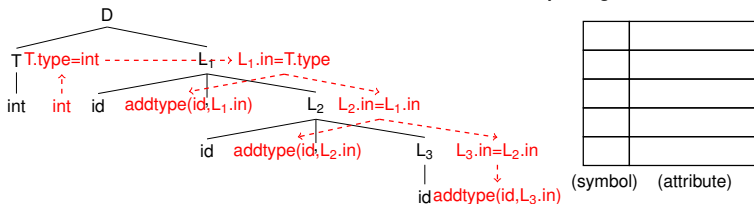
(state) (symbol) (attribute)

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



Syntax Translation Scheme for Top-Down Parsing

When using LL parsing (top-down parsing),

□ it is natural to evaluate inherited attributes

D

parsing stack:

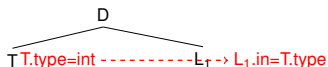
D	

(symbol) (attribute)

Syntax Translation Scheme for Top-Down Parsing

When using LL parsing (top-down parsing),

□ it is natural to evaluate inherited attributes



parsing stack:

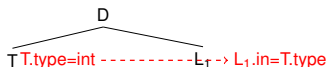
T	T.type=int
L ₁	L ₁ .in=()

(symbol) (attribute)

Syntax Translation Scheme for Top-Down Parsing

When using LL parsing (top-down parsing),

□ it is natural to evaluate inherited attributes



parsing stack:

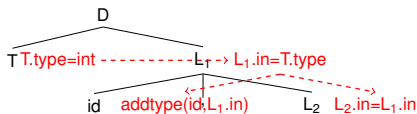
L_1	$L_1.in=int$

(symbol) (attribute)

Syntax Translation Scheme for Top-Down Parsing

When using LL parsing (top-down parsing),

it is natural to evaluate inherited attributes



parsing stack:

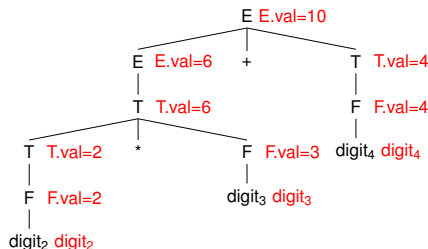
id	id.type=L ₁ .in
,	
L ₂	L ₂ .in=intL ₁ .in

(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:

(symbol) (attribute)

Syntax Translation Scheme for Top-Down Parsing

When using LL parsing (top-down parsing),

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

E

parsing stack:

E	

(symbol) (attribute)

Syntax Translation Scheme for Top-Down Parsing

When using LL parsing (top-down parsing),

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

E E.val=?

parsing stack:

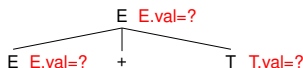
E	E.val=?

(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

$L \rightarrow E$	$\{ \text{print}(E.\text{val}) \}$
$E \rightarrow E_1 + T$	$\{ E.\text{val} = E_1.\text{val} + T.\text{val} \}$
$E \rightarrow T$	$\{ E.\text{val} = T.\text{val} \}$
$T \rightarrow T_1 + F$	$\{ T.\text{val} = T_1.\text{val} + F.\text{val} \}$
$T \rightarrow F$	$\{ T.\text{val} = F.\text{val} \}$
$F \rightarrow (E)$	$\{ F.\text{val} = E.\text{val} \}$
$F \rightarrow \text{digit}$	$\{ F.\text{val} = \text{digit}.\text{lexval} \}$

Example of Grammar Rewriting

$$E \rightarrow T \ R$$
$$R \rightarrow + \ T \ R$$
$$R \rightarrow - \ T \ R$$
$$R \rightarrow \varepsilon$$
$$T \rightarrow (\ E \)$$
$$T \rightarrow \text{num}$$

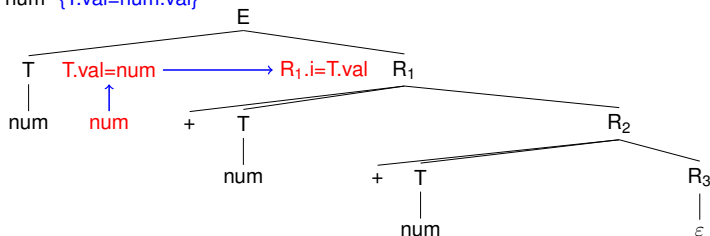
 Both inherited and synthesized attributes are used

- 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

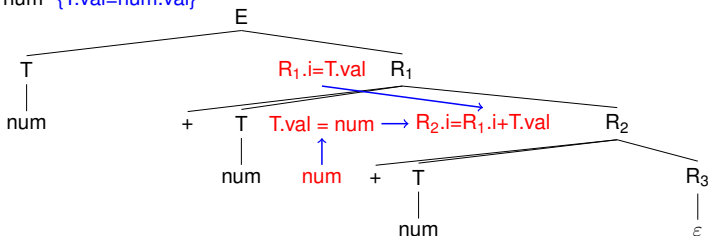
$E \rightarrow T \quad \{R.i=T.val\} \quad R \quad \{E.i=R.s\}$
 $R \rightarrow + \quad T \quad \{R.i=R.i+T.val\} \quad R_1 \quad \{R.s=R_1.s\}$
 $R \rightarrow - \quad \{R.i=R.i-T.val\} \quad R_1 \quad \{R.s=R_1.s\}$
 $R \rightarrow \varepsilon \quad \{R.s=R_1.s\}$
 $T \rightarrow (\quad E \quad \{T.val=E.val\}$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



Translation Scheme for the Example

- Evaluating synthesized attribute with the introduction of new inherited attribute

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



Translation Scheme for the Example

- Evaluating synthesized attribute with the introduction of new inherited attribute

$E \rightarrow T \quad \{R.i=T.val\} \quad R \quad \{E.i=R.s\}$

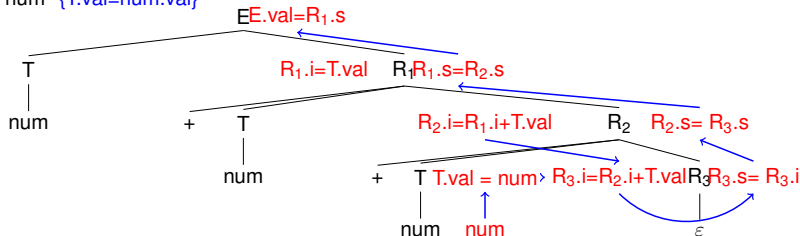
$R \rightarrow + \quad T \quad \{R.i=R.i+T.val\} \quad R_1 \quad \{R.s=R_1.s\}$

$R \rightarrow - \quad \{R.i=R.i-T.val\} \quad R_1 \quad \{R.s=R_1.s\}$

$R \rightarrow \varepsilon \quad \{R.s=R_1.s\}$

$T \rightarrow (\quad E \quad \{T.val=E.val\}$

$T \rightarrow \text{num} \quad \{T.val=\text{num.val}\}$



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



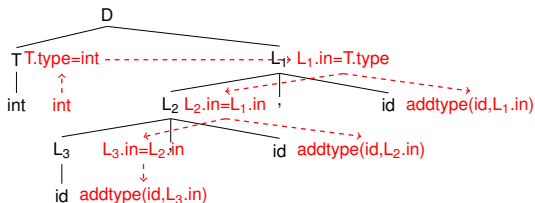
Solution: let us hack the stack to find the location

$$D \rightarrow T \ L$$

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

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

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

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


parsing stack:

$S_?$	\$	-
(state)	(symbol)	(attribute)

$$D \rightarrow T \quad L$$

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

T \rightarrow real {stack[top]=real}

$$L \rightarrow L, \text{ id } \{\text{addtype}(\text{stack}[\text{top}], \text{stack}[\text{top}-3])\}$$
$$L \rightarrow \text{id} \quad \{\text{addtype}(\text{stack}[\text{top}], \text{stack}[\text{top}-1])\}$$

int

9

id

9

id

id

parsing stack:

S ₇	\$	-

(state) (symbol) (attribute)

$$D \rightarrow T \ L$$

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

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

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

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

$$T \text{ T.type=int}$$

$$\begin{array}{c} | \\ \text{int} \end{array} \quad \begin{array}{c} \uparrow \\ \text{int} \end{array}$$

,

id

,

id

id

parsing stack:

S?	T	T.type=int
S?	\$	-

(state) (symbol) (attribute)

$$D \rightarrow T \ L$$

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

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

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

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

T $T.\text{type}=\text{int}$
 \downarrow \uparrow
 int int , id
 , id
 id $\text{addtype}(\text{id}, L_3.\text{in})$

parsing stack:

$S_?$	id	$\text{id.type}=\text{stack}[\text{top}-1]$
$S_?$	T	$T.\text{type}=\text{int}$
$S_?$	\$	-

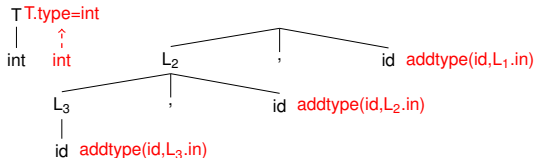
(state) (symbol) (attribute)

$$D \rightarrow T \ L$$

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

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

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

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


parsing stack:

S_7	id	id.type=stack[top-3]
S_7	,	
S_7	L_3	$L_3.\text{in} = \text{int}$
S_7	T	$T.\text{type} = \text{int}$
S_7	\$	-

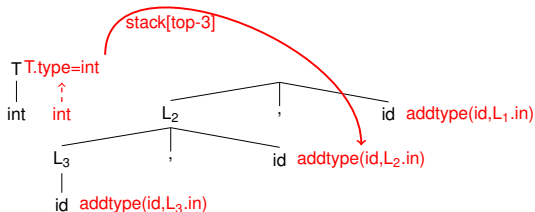
(state) (symbol) (attribute)

$$D \rightarrow T \ L$$

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

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

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

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


parsing stack:

S_7	id	id.type=stack[top-3]
S_7	,	
S_7	L_3	$L_3.\text{in}=\text{int}$
S_7	T	$T.\text{type}=\text{int}$
S_7	\$	-

(state) (symbol) (attribute)

Marker

□ Adding “ $M \rightarrow \varepsilon$ ” to assist evaluation

convert

$A \rightarrow X \{\text{rule}\} Y$

to

$A \rightarrow X M Y$

$M \rightarrow \varepsilon \{\text{rule}\}$

- 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:

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

Solution:

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

That is:

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

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:

$S \rightarrow a A B C E D$

$S \rightarrow b A F B C F D$

$C \rightarrow c \text{ /* } C.s = f(A.s) \text{ */}$

$D \rightarrow d \text{ /* } D.s = f(B.s) \text{ */}$

Answer

$S \rightarrow a A B C E D$

$S \rightarrow b A D M B C F D$

$C \rightarrow c \text{ /* } C.s = f(\text{stack}[\text{top}-2]) \text{ */}$

$D \rightarrow d \text{ /* } D.s = f(\text{stack}[\text{top}-3]) \text{ */}$

$M \rightarrow \varepsilon \text{ /* } M.s = f(\text{stack}[\text{top}-2]) \text{ */}$

 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