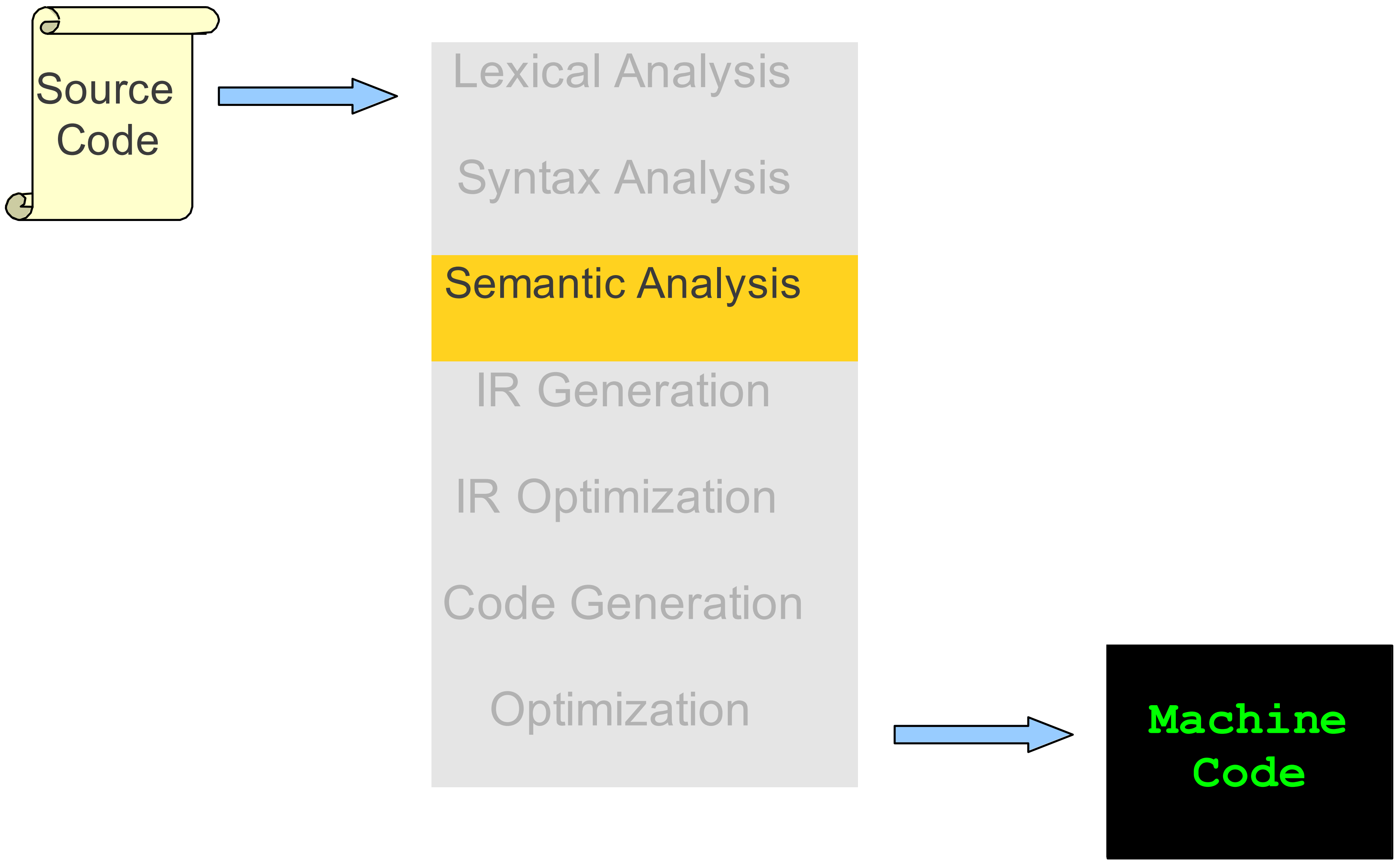
Semantic Analysis

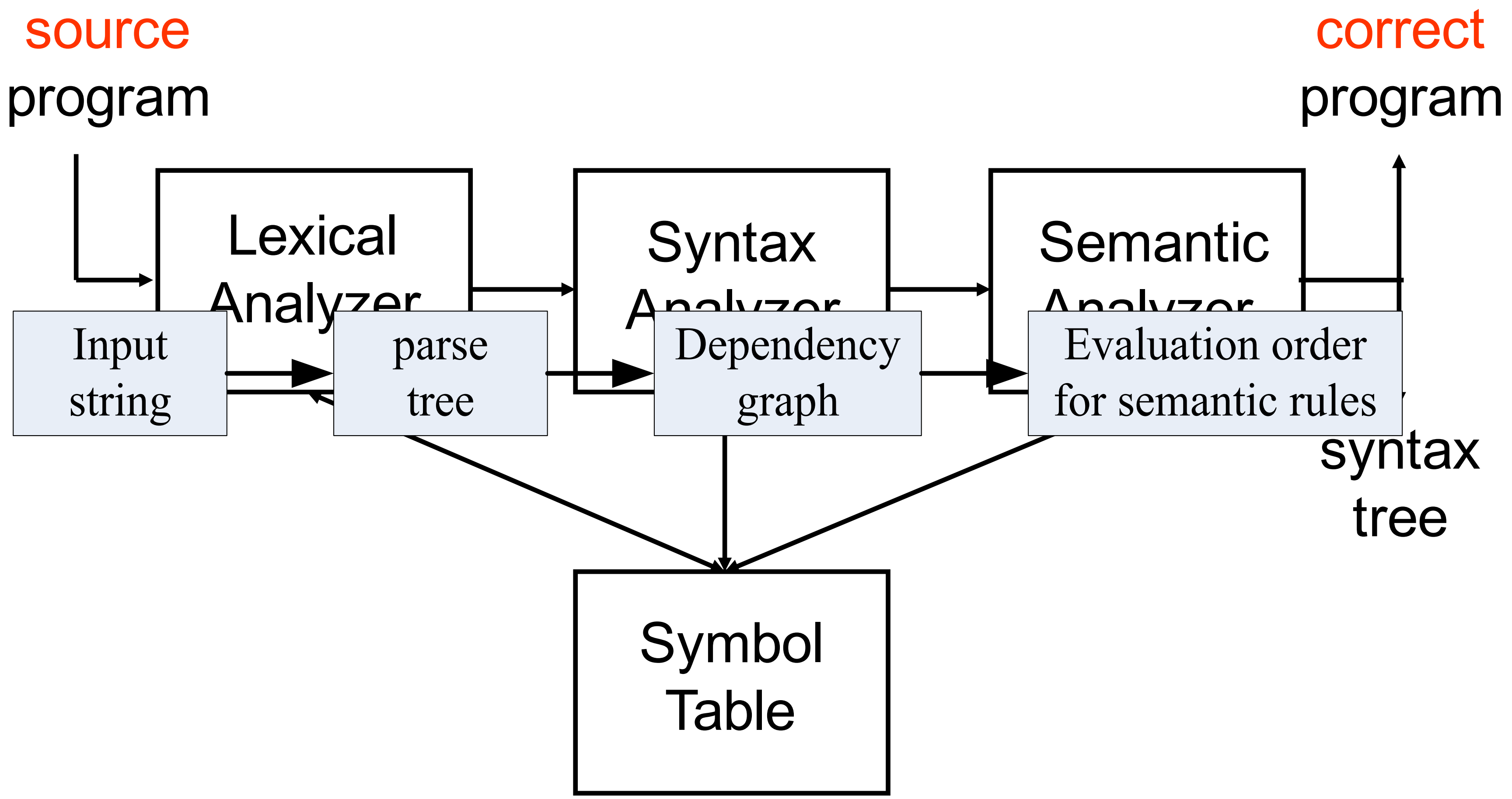
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where are we

- Program is *lexically* well-formed:
  - Identifiers have valid names. Strings
  - are properly terminated. No stray
  - characters.
- Program is *syntactically* well-formed:
  - Class declarations have the correct structure.
  - Expressions are syntactically valid.
- Does this mean that the program is legal?

Semantic Analyzer



A Short Program

```
class MyClass implements MyInterface {


    string myInteger;

    void doSomething() {
        int[] x = new string  ;

        x[5] = myInteger  *  y  ;
    }

    void doSomething() {


    }

   int fibonacci(int n) {
   return doSomething() + fibonacci(n-1);
    }
}
```

A Short Program

```
class MyClass implements MyInterface {

    string myInteger;

    void doSomething() {
        int[] x = new string ;
        x[5] = myInteger * y ;
    }

    void doSomething() {

    }

    int fibonacci(int n) {
    return doSomething() + fibonacci(n-1);
    }
}
```

Interface not declared

Wrong Type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

Keywords

- Semantic Analyzer
- Attributes
  - Synthesized Attributes
  - Inherited Attributes
- Attribute Grammars
- Syntax Directed Definition (SDD)
  - S-Attributes Definition
  - L-Attributes Definition
- Dependency Graph
- Syntax-Directed Translation Schemes

Semantic Analysis

- Ensure that the program has a well-defined **meaning**.
- Verify properties of the program that aren't caught during the earlier phases:
  - Variables are declared before they're used. Expressions have the right types.
  - Arrays can only be instantiated with `NewArray`. Classes don't inherit from nonexistent base classes
  - …
- Once we finish semantic analysis, we know that the user's input program is legal.

Other Goals of Semantic Analysis

- Gather useful information about program for later phases:
  - Determine what variables are meant by each identifier.
  - Build an internal representation of inheritance hierarchies.
  - Count how many variables are in scope at each point.

Attribute Grammars

- Each grammar symbol is associated with a set of semantic attributes
- Each production is associated with a set of semantic rules for computing semantic attributes
- An attribute grammar is a context free grammar with associated semantic attributes and semantic rules

Attribute Grammar (cont.)

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends* on attributes $c_1,c_2,\ldots,c_n$.
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.
- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

Example 5.1

Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Syntax-Directed Translation

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:

  may generate intermediate codes

  may put information into the symbol table

  may perform type checking

  may issue error messages

  may perform some other activities

  in fact, they may perform almost any activities.
- An attribute may hold almost any thing.

  a string, a number, a memory location, a complex record.

Syntax-Directed Definitions and Translation Schemes

- When we associate semantic rules with productions, we use two notations:

  Syntax-Directed Definitions

  Translation Schemes

- **Syntax-Directed Definitions:**

  give high-level specifications for translations

  hide many implementation details such as order of evaluation of semantic actions.

  We associate a production rule with a set of semantic actions, and we do not say ***when*** they will be evaluated.

- **Translation Schemes:**

  indicate the order of evaluation of semantic actions associated with a production rule.

  In other words, translation schemes give a little bit information about implementation details.

Syntax-Directed Definitions

Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

Translation Schemes

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

Syntax-Directed Definitions

- A syntax-directed definition is a generalization of a context-free grammar in which:

  Each grammar symbol is associated with a set of attributes.

  This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited attributes** of that grammar symbol.

  Each production rule is associated with a set of semantic rules.

- ***Semantic rules*** set up dependencies between attributes which can be represented by a ***dependency graph***.
- This *dependency graph* determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Form of a Syntax-Directed Definition (Semantic Rules)

- In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

  $b=f(c_1,c_2,\ldots,c_n)$ where *f* is a function,

  and *b* can be one of the followings:

  ➔ *b* is a **synthesized attribute** of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

  OR

  ➔ *b* is an **inherited attribute** of one of the grammar symbols in α (on the right side of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Syntax-Directed Definition – Example5.1

Production	**Semantic Rules**
L → E **return**	print(E.val)
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = **digit**.lexval

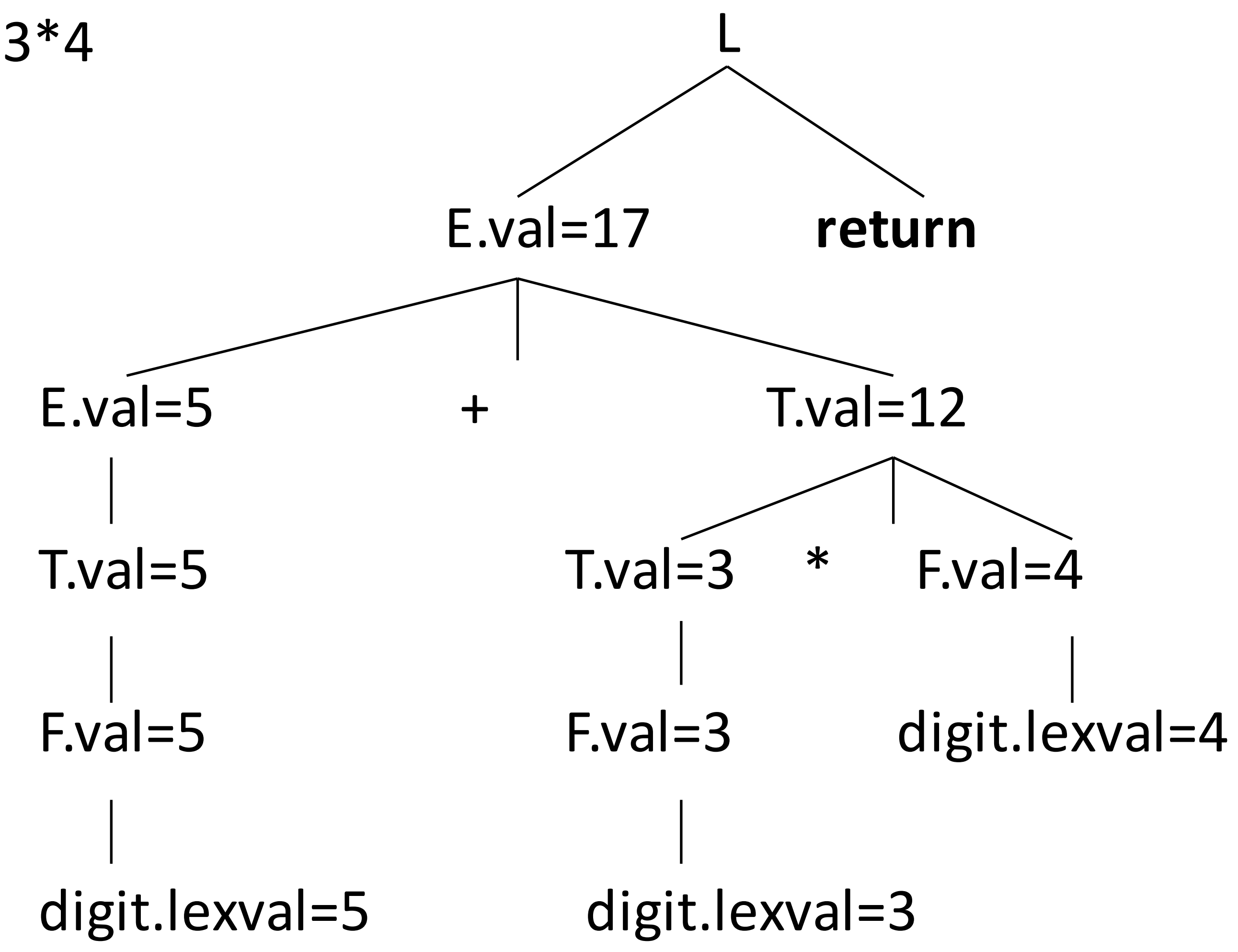
- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Annotated Parse-Trees

- Parse-tree that also shows the values of the attributes at each node.
- Attributes in the leaves of the annotated parse-tree are determined by the lexical analyzer.
- Values of attributes in inner nodes of annotated parse-tree are determined by the semantic-rules.
- If a syntax-directed definition employs only *Synthesized* attributes the evaluation of all attributes can be done in a bottom-up fashion.
- *Inherited* attributes would require more arbitrary “traversals” of the annotated parse-tree.
- A ***dependency graph*** suggests possible *evaluation orders* for an annotated parse-tree.

Annotated Parse Tree -- Example

Input: 5+3*4



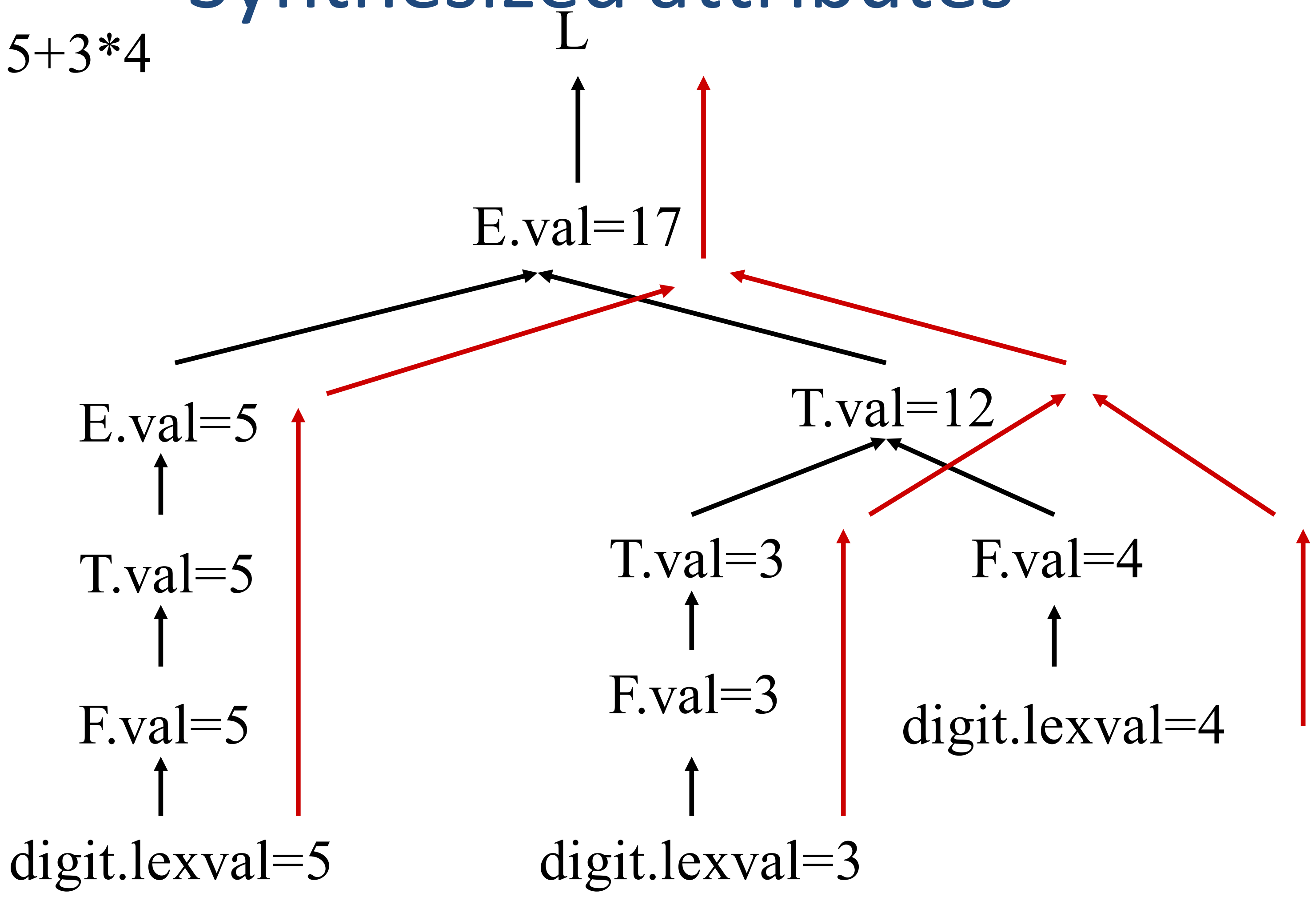
Annotated parse tree for 5+3*4

Synthesized Attributes

$L \rightarrow E$ return	print(E.val)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **digit**	F.val := **digit**.lexval

Synthesized attributes

Input: 5+3*4



S-attributed definition can be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the leaves to the root.

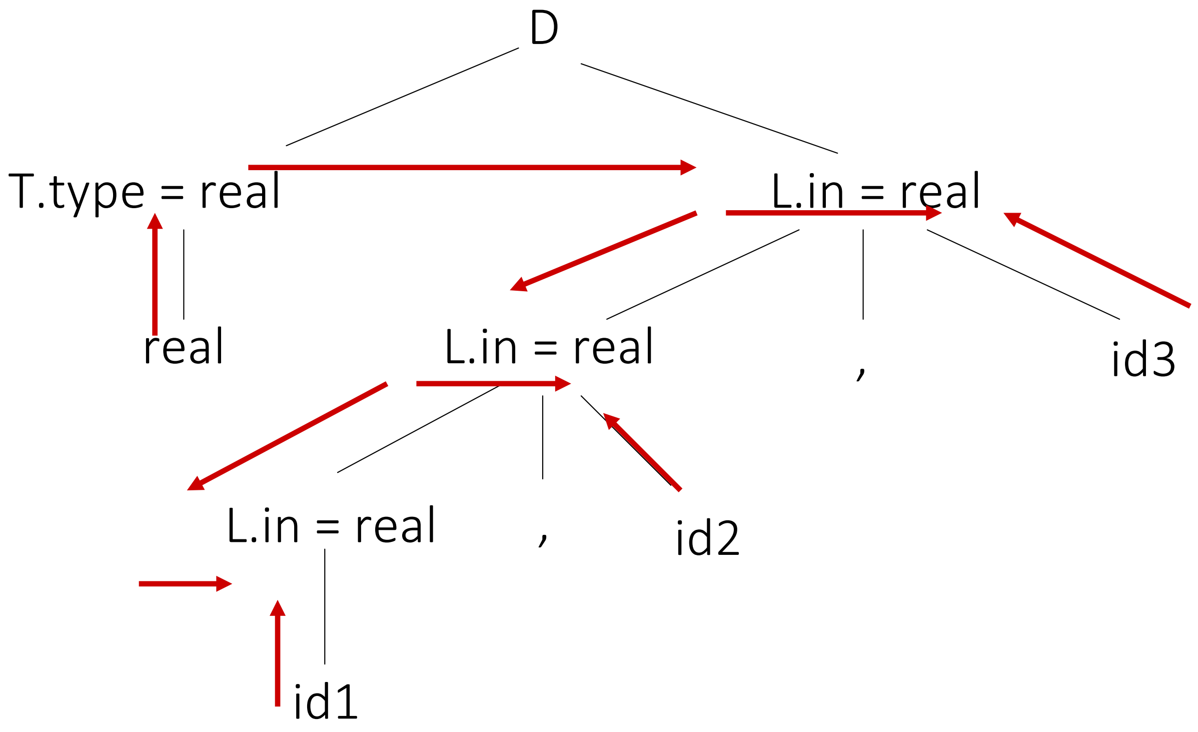
Inherited Attributes(Example 5.10)

Production	**Semantic Rules**
$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = real
$L \rightarrow L_1$, **id**	L_1.inh = L.inh, addtype(**id**.entry,L.inh)
$L \rightarrow$ **id**	addtype(**id**.entry,L.inh)

Symbol T is associated with a synthesized attribute *type*.
Symbol L is associated with an inherited attribute *in.*

Production	**Semantic Rules**
D → T L	L.inh = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L1 , **id**	L1.inh = L.inh, addtype(**id**.entry,L.inh)
L → **id**	addtype(**id**.entry,L.inh)

string : real id$_1$, id$_2$, id$_3$



parse tree with inherited attribute in at each node labeled L

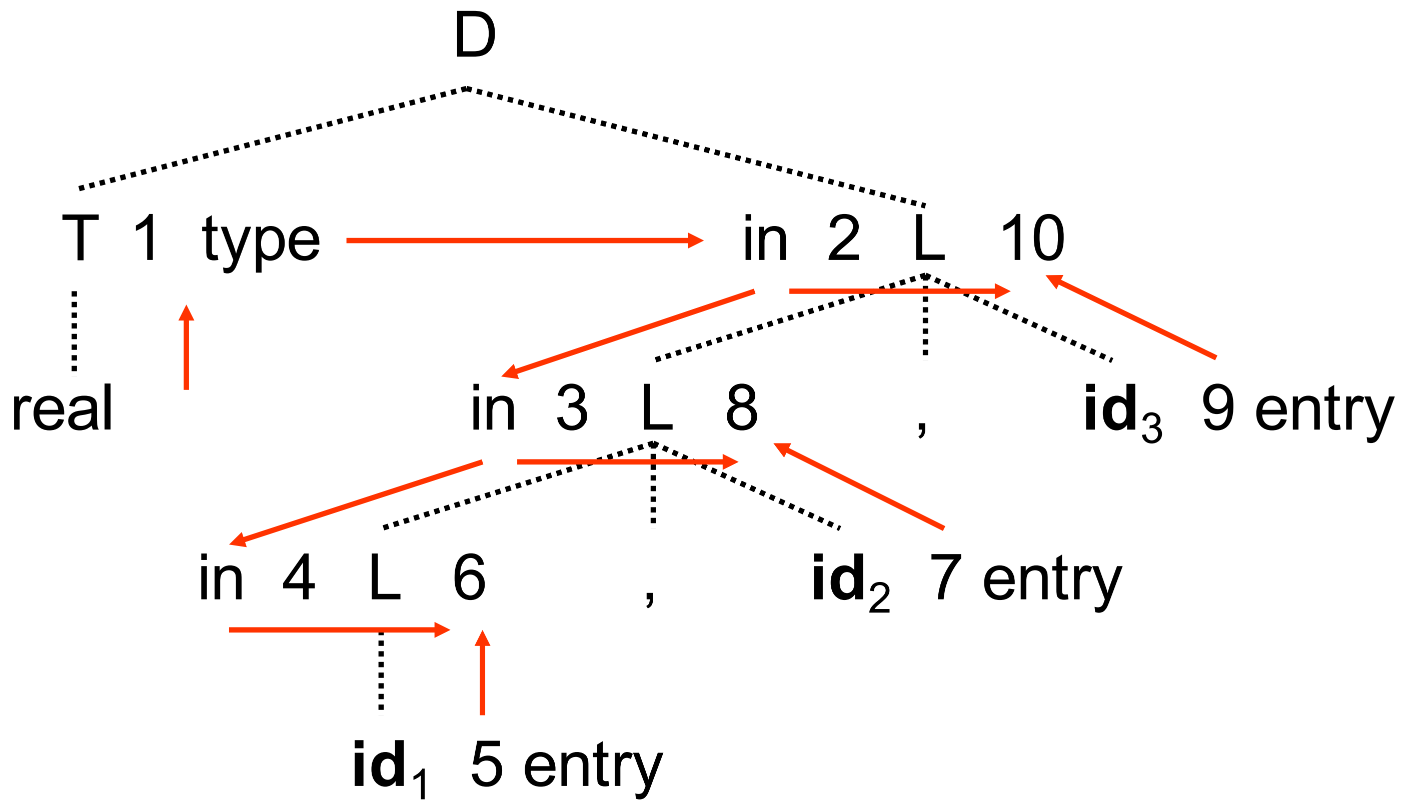
Dependencies of Attributes

- In the semantic rule
  $$b := f(c_1, c_2, \ldots, c_k)$$
  we say b depends on $c_1, c_2, \ldots, c_k$

- The semantic rule for b must be evaluated after the semantic rules for $c_1, c_2, \ldots, c_k$

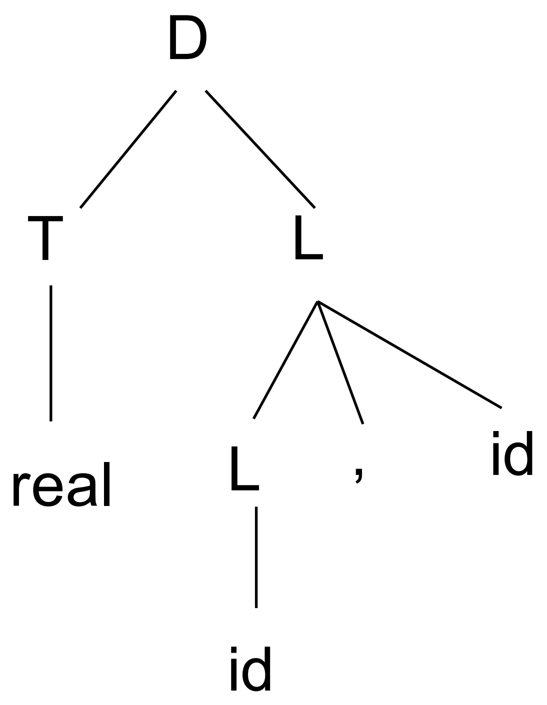
- The dependencies of attributes can be represented by a directed graph called dependency graph

Dependency Graphs – Inherited Attributes

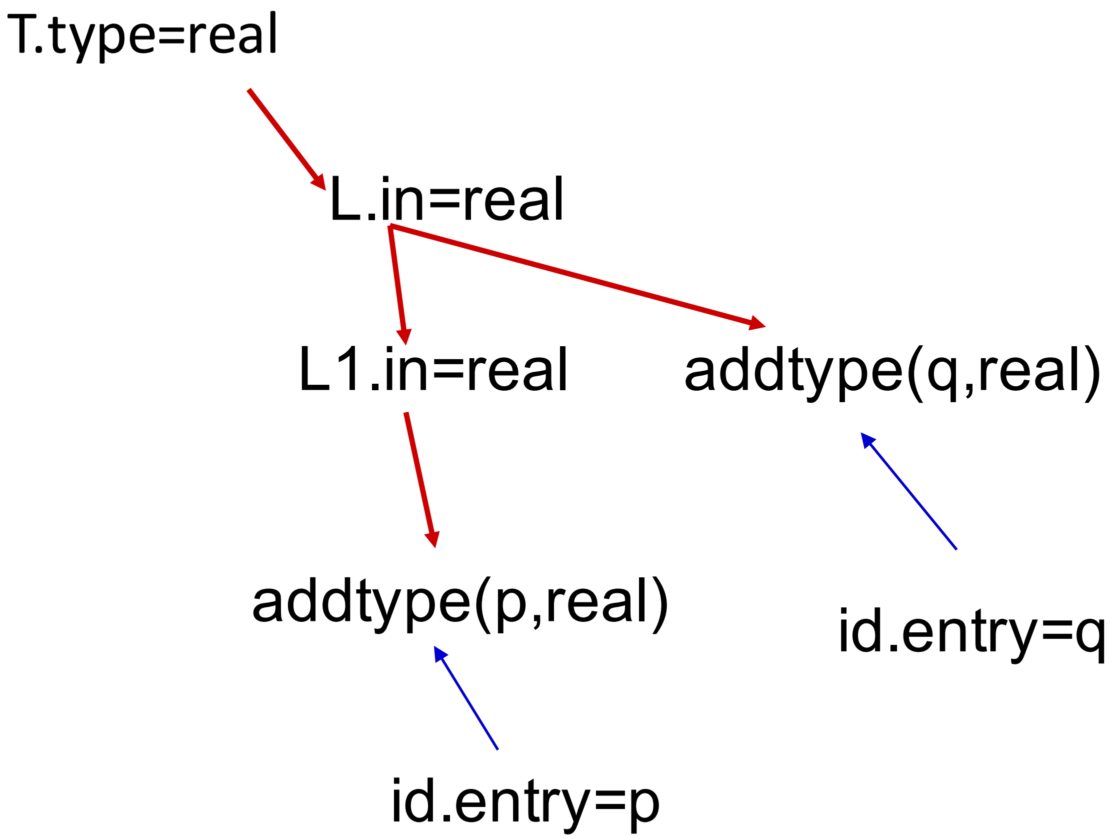


Dependency Graph (cont.)

Input: real p, q



parse tree



dependency graph

S-Attributed Definitions & L-Attributed Definitions

- Syntax-directed definitions are used to specify syntax-directed translations.
- To create a translator for an arbitrary syntax-directed definition can be difficult.
- We would like to evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).
- We will look at two sub-classes of the syntax-directed definitions:
  - **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
  - **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.
- To implement S-Attributed Definitions and L-Attributed Definitions are easy (we can evaluate semantic rules in a single pass during the parsing).

S-Attributed Attribute Grammars

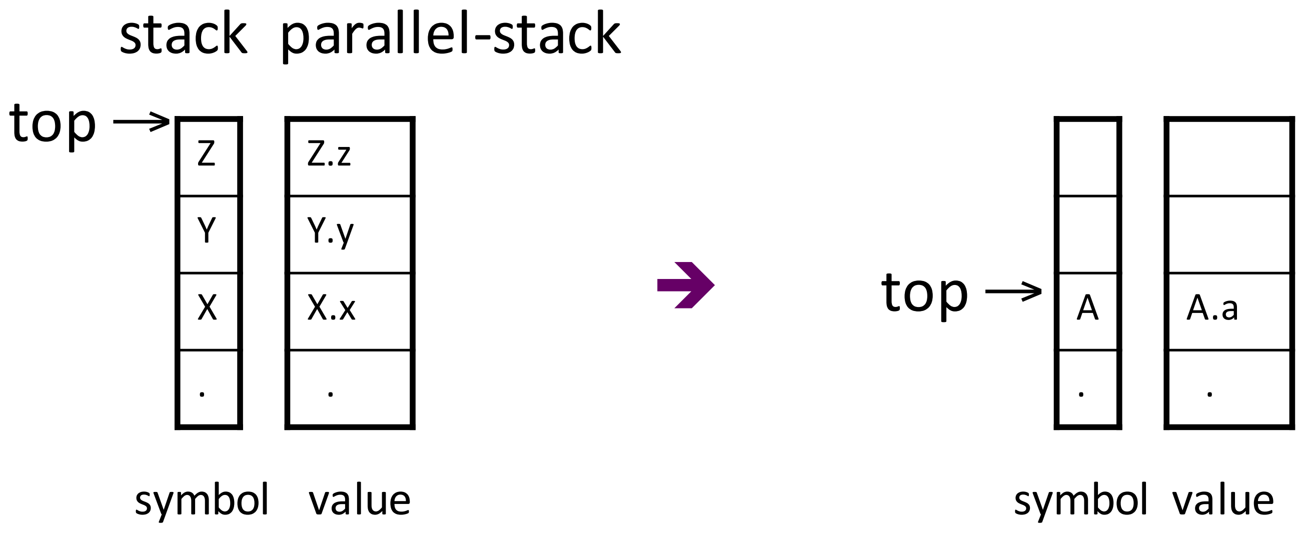
- An attribute grammar is **S-attributed** if it uses synthesized attributes exclusively

An example:

$L \rightarrow E$ return	print(E.val)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **digit**	F.val := **digit**.lexval

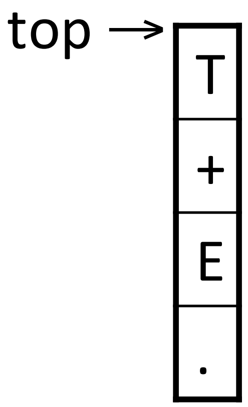
Bottom-Up Evaluation of S-Attributed Definitions

- S-attributed Definition: Syntax-Directed Definition using only Synthesized attributes.
- Stack of a LR(1) parser contains states.
- Recall that each state corresponds to some grammar symbol (and many different states might correspond to the same grammar symbol)
- Extend stack entries to include the attribute(s).
- Modify stack contents as productions are selected.
- A → XYZ A.a=f(X.x,Y.y,Z.z) where all attributes are synthesized.



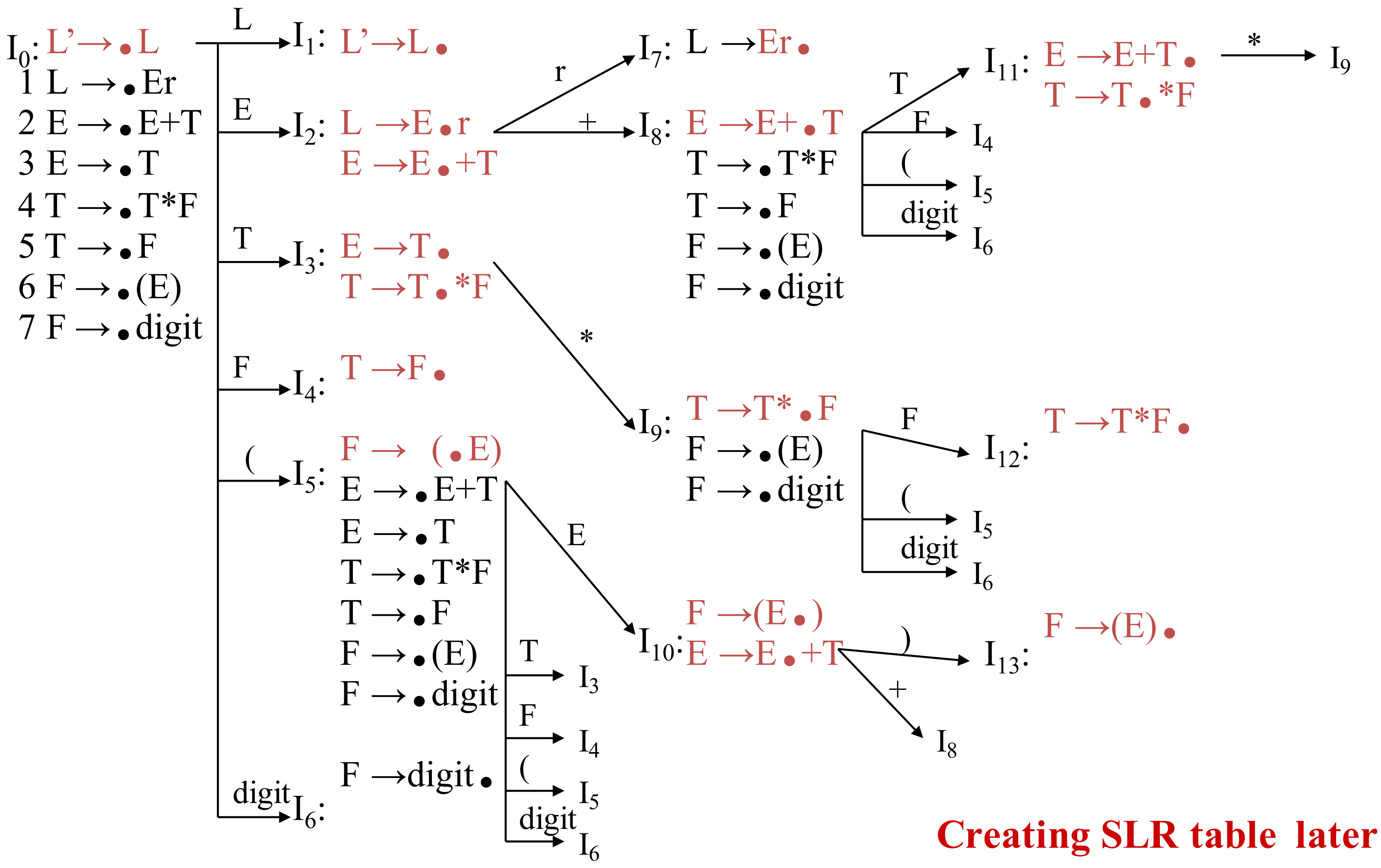
Bottom-Up Eval. of S-Attributed Definitions (cont.)

Production	**Semantic Rules**
L → E **return**	print(val[top-1])
E → E1 + T	val[ntop] = val[top-2] + val[top]
E → T	
T → T1 * F	val[ntop] = val[top-2] * val[top]
T → F	
F → (E)	val[ntop] = val[top-1]
F → **digit**	



- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Canonical LR(0) Collection for The Grammar



Creating SLR table later

SLR(1) parsing table

states	Action table							Goto table			
	+	*	(	)	digit	r	$	L	E	T	F
0			S5		S6			1	2	3	4
1							acc				
2	S8					S7					
3	R3	S9		R3							
4	R5	R5		R5							
5			S5		S6				10	3	4
6	R7	R7		R7							
7							R1				
8			S5		S6					11	4
9			S5		S6						12
10	S8			S13							
11	R2	S9		R2							
12	R4	R4		R4							
13	R6	R6		R6							

Bottom-Up Evaluation

At each shift of **digit**, we also push **digit.lexval** into *val-stack*.

stack	val-stack	input	action	semantic rule
0		5+3*4r	s6	digit.lexval(5) into val-stack
0digit6	5	+3*4r	F→digit	F.val=d.lexval – do nothing
0F4	5	+3*4r	T→F	T.val=F.val – do nothing
0T3	5	+3*4r	E→T	E.val=T.val – do nothing
0E2	5	+3*4r	s8	push empty slot into val-stack
0E2+8	5-	3*4r	s6	digit.lexval(3) into val-stack
0E2+8digit6	5-3	*4r	F→digit	F.val=digit.lexval – do nothing
0E2+8F4	5-3	*4r	T→F	T.val=F.val – do nothing
0E2+8T11	5-3	*4r	s9	push empty slot into val-stack
0E2+8T11*9	5-3-	4r	s6	digit.lexval(4) into val-stack
0E2+8T11*9digit6	5-3-4	r	F→digit	F.val=digit.lexval – do nothing
0E2+8T11*9F12	5-3-4	r	T→T*F	$T.val=T_1.val*F.val$
0E2+8T11	5-12	r	E→E+T	$E.val=E_1.val+T.val$
0E2	17	r	s7	push empty slot into val-stack
0E2r7	17-	$	L→Er	print(17), pop empty slot from val-stack
0L1	17	$	acc	

L-Attributed Definitions

- S-Attributed Definitions can be efficiently implemented.
- We are looking for a larger (larger than S-Attributed Definitions) subset of syntax-directed definitions which can be efficiently evaluated.

  → **L-Attributed Definitions**

- L-Attributed Definitions can always be evaluated by the depth first visit of the parse tree.
- This means that they can also be evaluated during the parsing.

L-Attributed Attribute Grammars

- An attribute grammar is **L-attributed** if each attribute computed in each semantic rule for each production

  $$A \rightarrow X_1\ X_2\ \ldots\ X_{j-1}\ X_j\ \ldots\ X_n$$

  is a synthesized attribute, or an inherited attribute of X_j, $1 \leq j \leq n$, depending only on

  1. the attributes of $X_1, X_2, \ldots, X_{j-1}$
  2. the inherited attributes of A

An Example

$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := integer
$T \rightarrow$ **float**	T.type := float
$L \rightarrow L_1$, **id**	L_1.in := L.in;
	addtype(**id**.entry, L.in)
$L \rightarrow$ **id**	addtype(**id**.entry, L.in)

A Definition which is NOT L-Attributed

Productions	**Semantic Rules**
$A \rightarrow L\ M$	L.in=l(A.i), M.in=m(L.s), A.s=f(M.s)
$A \rightarrow Q\ R$	R.in=r(A.in), Q.in=q(R.s), A.s=f(Q.s)

- This syntax-directed definition is not L-attributed because the semantic rule Q.in=q(R.s) violates the restrictions of L-attributed definitions.
- When Q.in must be evaluated before we enter to Q because it is an inherited attribute.
- But the value of Q.in depends on R.s which will be available after we return from R. So, we are not be able to evaluate the value of Q.in before we enter to Q.

Translation Schemes

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated?).

- A **translation scheme** is a context-free grammar in which:
  - **attributes are associated with the grammar symbols and**
  - **semantic actions enclosed between braces {} are inserted within the right sides of productions.**

- *Ex:* A → { ... } X { ... } Y { ... }

- Semantic Actions

Translation Schemes (cont.)

- When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
- These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
- In translation schemes, we use ***semantic action*** terminology instead of ***semantic rule*** terminology used in syntax-directed definitions.
- The position of the semantic action on the right side indicates ***when*** that semantic action will be evaluated.

Translation Schemes (cont.)

- A CFG with "semantic actions" *embedded* into its productions. Useful for binding the order of evaluation into the parse-tree.
- As before semantic actions might refer to the attributes of the symbols.
- Example.

  expr → expr + term { *print*("+") }

  expr → expr - term { *print*("-") }

  expr → term

  term → 0 { *print*("0") }

  …

  term → 9 { *print*("9") }

- **=>**Traversing a parse tree for a Translation-Scheme produces the translation. (we will employ only DFS)
- =>Translation schemes also help in materializing L-attributed Definitions.

Translation Schemes - example

The production and semantic rule:

Production	semantic rule
$T \longrightarrow T1 * F$	T.val := T1.val * F.val

Yield the following production and semantic action:

$T \longrightarrow T1 * F$ {T.val := T1.val * F.val}

Writing Translation Schemes

- Start with a Syntax-Directed Definition.
- *In General*: Make sure that we never refer to an attribute that has not been defined already.
- For S-Attributed Definitions, we simply put all semantic rules into {…} at the rightmost of each production.
- If both inherited and synthesized attributes are involved:
  - An **inherited attribute** for a symbol on the **RHS** of a production must be computed in an action before that symbol.
  - An action must not refer to a synthesized attribute of a symbol that is to the right.
  - A **synthesized attribute** for the NT on the LHS can only be computed after all attributes it references are already computed. (the action for such attributes is typically placed in the rightmost end of the production).
- L-attributed definitions are suited for the above…

Examples

$\mathbf{S \rightarrow A_1 \{S.\mathit{s} = A_1.\mathit{s}+A_2.\mathit{s}\}\ A_2}$

$\mathbf{A \rightarrow}$ **a** $\mathbf{\{A.\mathit{s} = 1\}}$

This will not work…

On the other hand this is good:

$\mathbf{S \rightarrow A_1\ A_2\ \{S.\mathit{s} = A_1.\mathit{s}+A_2.\mathit{s}\}}$

$\mathbf{A \rightarrow}$ **a** $\mathbf{\{A.\mathit{s} = 1\}}$

Examples II

$\mathbf{S \rightarrow A_1\ A_2\ \{A_1.\mathit{in} = 1;\ A_2.\mathit{in} = 2\}}$

$\mathbf{A \rightarrow a\ \{\mathit{print}(A.\mathit{in})\}}$

This will not work…

On the other hand this is good:

$\mathbf{S \rightarrow \{A_1.\mathit{in} = 1;\ A_2.\mathit{in} = 2\}\ A_1\ A_2}$

$\mathbf{A \rightarrow a\ \{\mathit{print}(A.\mathit{in})\}}$

Or even:

$\mathbf{S \rightarrow \{A_1.\mathit{in} = 1\}\ A_1\ \{A_2.\mathit{in} = 2\}\ A_2}$

$\mathbf{A \rightarrow a\ \{\mathit{print}(A.\mathit{in})\}}$

Translation Schemes for S-attributed Definitions

- If the syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

<u>Production</u>	<u>Semantic Rule</u>	
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$	➔ a production of a syntax directed definition

$$\Downarrow$$

$E \rightarrow E_1 + T \; \{ \; E.val = E_1.val + T.val \; \}$ ➔ the production of the corresponding translation scheme

Inherited Attributes in Translation Schemes

- If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules:
  - An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.
  - A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
  - A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed (we normally put this semantic action at the end of the right side of the production).

- With a L-attributed syntax-directed definition, it is always possible to construct a corresponding translation scheme which satisfies these three conditions (This may not be possible for a general syntax-directed translation).

An example

- The following translation scheme does not satisfy these three requirements:

```
S -> A1  A2       {A1.in = 1; A2.in = 2}
A -> a   {print (A.in)}
```

A Translation Scheme Example

- A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

E → T R

R → + T { print("+") } R1

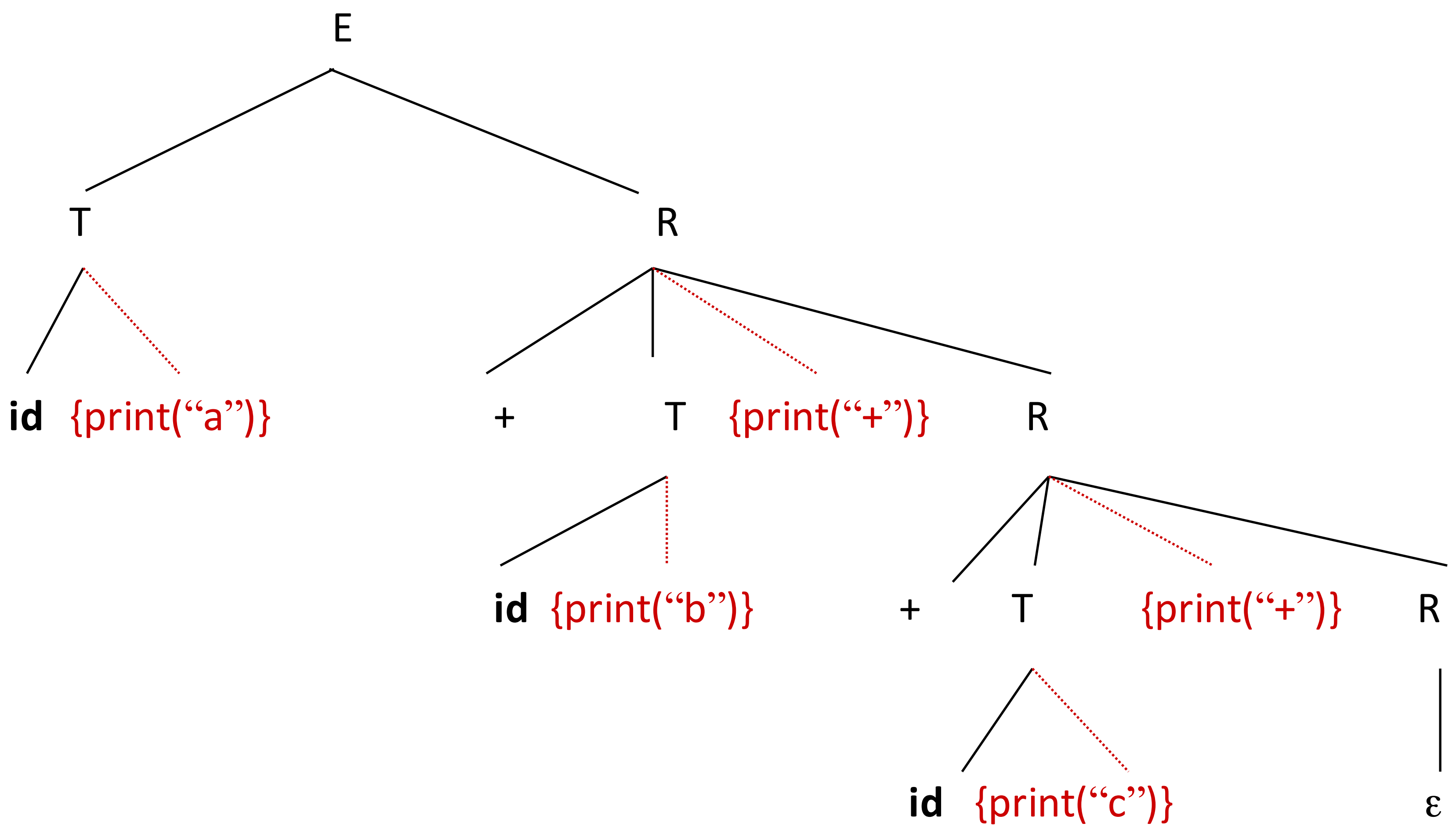
R → ε

T → **id** { print(**id**.name) }

What kind of expression?

$$\begin{tikzcd}
\text{a+b+c} \arrow[r] & \text{ab+c+} \\
\text{infix expression} \arrow[u] & \text{postfix expression} \arrow[u]
\end{tikzcd}$$

A Translation Scheme Example (cont.)



- The depth first traversal of the parse tree (executing the semantic actions in that order) will produce the postfix representation of the infix expression.

Derivation with semantic actions

E → T R
R → + T { print("+") } R1
R → ε
T → **id** { print(**id**.name) }

Left-most derivation:

E ⇒T R
⇒ **id** { print(**id**.name) } R
⇒ **id** { print(**id**.name) } + T { print("+") } R
⇒ **id** { print(**id**.name) } + **id** { print(**id**.name) } { print("+") } R
⇒ **id** { print(**id**.name) } + **id** { print(**id**.name) } { print("+") } + T { print("+") } R
⇒ **id** { print(**id**.name) } + **id** { print(**id**.name) } { print("+") } + **id** { print(**id**.name) }
{ print("+") } R
⇒ **id** { print(**id**.name) } + **id** { print(**id**.name) } { print("+") } + **id** { print(**id**.name) }
{ print("+") } ε

How right-most derivation?
Do it by yourself.

Top-Down Translation

- We will look at the implementation of L-attributed definitions during predictive parsing.
- Instead of the syntax-directed translations, we will work with translation schemes.
- We will see how to evaluate inherited attributes (in L-attributed definitions) during recursive predictive parsing.
- We will also look at what happens to attributes during the left-recursion elimination in the left-recursive grammars.

Top-Down Translation (cont.)

- For each nonterminal,
  - –inherited attributes → formal parameters
  - –synthesized attributes → returned values
- For each production,
  - –for each terminal X with synthesized attribute x,
    save X.x; match(X);
  - –for nonterminal B, c := B(b_1, b_2, …, b_k);
  - –for each semantic rule, copy the rule to the parser

A Translation Scheme with Inherited Attributes

D → T **id**{ addtype(**id**.entry,T.type), L.in = T.type } L

T → **int** { T.type = integer }

T → **real** { T.type = real }

L → **id** { addtype(**id**.entry,L.in), L_1.in = L.in } L_1

L → ε

This is a translation scheme for an L-attributed definitions.

Predictive Parsing (of Inherited Attributes)

```
procedure D() {
     int Ttype,  Lin,  identry;
     call T( &Ttype );  consume( id,  &identry );
     addtype( identry,  Ttype );  Lin=Ttype;
     call L(Lin);                                        a synthesized attribute (an output parameter)
}
procedure T( int *Ttype ) {
     if (currtoken is int) { consume(int); *Ttype=TYPEINT;  }
     else if (currtoken is real) { consume(real);  *Ttype=TYPEREAL;  }
     else { error("unexpected type"); }
                                                         an inherited attribute (an input parameter)
}
procedure L( int Lin ) {
     if (currtoken is id) { int L1in,  identry; consume( id,  &identry );
                              addtype( identry,  Lin ); L1in=Lin; call L( L1in ); }
     else if (currtoken is endmarker) { }
     else { error("unexpected token"); }
}
```

Eliminating Left Recursion from Translation Scheme

- A translation scheme with a left recursive grammar.

$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val \}$
$E \rightarrow E_1 - T$	$\{ E.val = E_1.val - T.val \}$
$E \rightarrow T$	$\{ E.val = T.val \}$
$T \rightarrow T_1 * F$	$\{ T.val = T_1.val * F.val \}$
$T \rightarrow F$	$\{ T.val = F.val \}$
$F \rightarrow (E)$	$\{ F.val = E.val \}$
$F \rightarrow$ **digit**	$\{ F.val =$ **digit**$.lexval \}$

- When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions

Eliminating Left Recursion (cont.)

$E \rightarrow T$ { E'.in=T.val } E' { E.val=E'.syn }

$E' \rightarrow + T$ { E'_1.in=E'.in+T.val } E'_1 { E'.syn = E'_1.syn}

$E' \rightarrow - T$ { E'_1.in=E'.in-T.val } E'_1 { E'.syn = E'_1.syn}

$E' \rightarrow \varepsilon$ { E'.syn = E'.in }

$T \rightarrow F$ { T'.in=F.val } T' { T.val=T'.syn }

$T' \rightarrow * F$ { T'_1.in=T'.in*F.val } T'_1 { T'.syn = T'_1.syn}

$T' \rightarrow \varepsilon$ { T'.syn = T'.in }

$F \rightarrow (E)$ { F.val = E.val }

$F \rightarrow$ **digit** { F.val = **digit**.lexval }

E → E1 + T	{ E.val = E1.val + T.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 }

Left-most dereivation with semantic rules

E → T { E’.in=T.val } E’ { E.val=E’.syn }

→ F { T’.in=F.val } T’ { T.val=T’.syn } { E’.in=T.val } E’ { E.val=E’.syn }

→ **digit** { F.val = **digit**.lexval } { T’.in=F.val } T’ { T.val=T’.syn } { E’.in=T.val } E’ { E.val=E’.syn }

→ **digit** { F.val = **digit**.lexval } { T’.in=F.val } ε { T’.syn = T’.in }{ T.val=T’.syn } { E’.in=T.val } E’ { E.val=E’.syn }

→ **digit** { F.val = **digit**.lexval } { T’.in=F.val } ε { T’.syn = T’.in }{ T.val=T’.syn } { E’.in=T.val } - T { E’1.in=E’.in-T.val } E’1 { E’.syn = E’1.syn}{ E.val=E’.syn }

→ **digit** { F.val = **digit**.lexval } { T’.in=F.val } ε { T’.syn = T’.in }{ T.val=T’.syn } { E’.in=T.val } - F { T’.in=F.val } T’ { T.val=T’.syn } { E’1.in=E’.in-T.val } E’1 { E’.syn = E’1.syn}{ E.val=E’.syn }

→ **digit** { F.val = **digit**.lexval } { T’.in=F.val } ε { T’.syn = T’.in }{ T.val=T’.syn } { E’.in=T.val } - **digit** { F.val = **digit**.lexval } { T’.in=F.val } T’ { T.val=T’.syn } { E’1.in=E’.in-T.val } E’1 { E’.syn = E’1.syn}{ E.val=E’.syn }

→ **digit** { F.val = **digit**.lexval } { T’.in=F.val } ε { T’.syn = T’.in }{ T.val=T’.syn } { E’.in=T.val } - **digit** { F.val = **digit**.lexval } { T’.in=F.val } ε { T’.syn = T’.in }{ T.val=T’.syn } { E’1.in=E’.in-T.val } E’1 { E’.syn = E’1.syn}{ E.val=E’.syn }

→ **digit** { F.val = **digit**.lexval } { T’.in=F.val } ε { T’.syn = T’.in }{ T.val=T’.syn } { E’.in=T.val } - **digit** { F.val = **digit**.lexval } { T’.in=F.val } ε { T’.syn = T’.in }{ T.val=T’.syn } { E’1.in=E’.in-T.val } ε { E’.syn = E’.in } { E’.syn = E’1.syn}{ E.val=E’.syn }

Eliminating Left Recursion (in general)

$A \rightarrow A_1\ Y$ { $A.a = g(A_1.a, Y.y)$ } a left recursive grammar with

$A \rightarrow X$ { $A.a=f(X.x)$ } synthesized attributes (a,y,x).

⇓ eliminate left recursion

inherited attribute of the new non-terminal

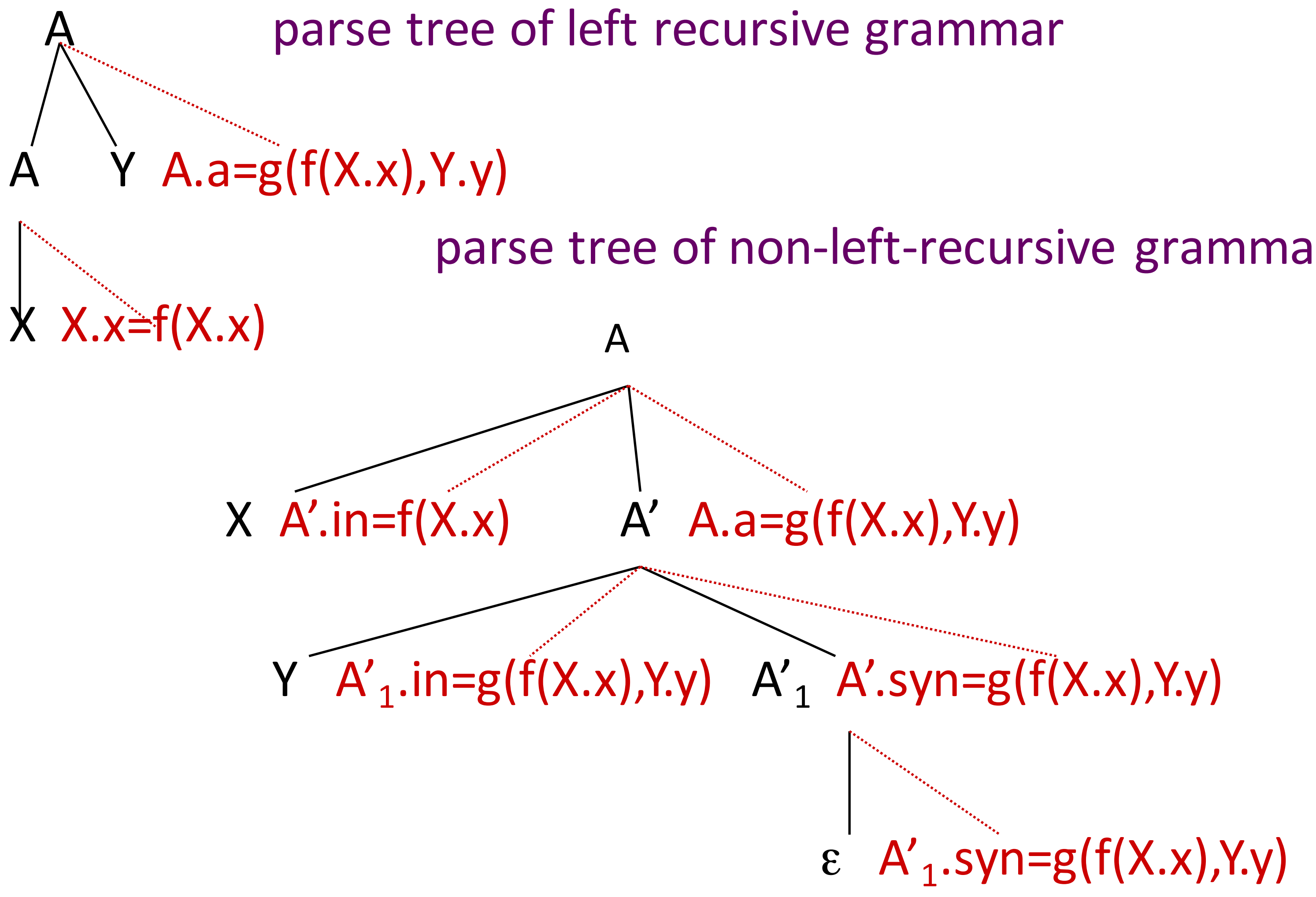
synthesized attribute of the new non-terminal

$A \rightarrow X$ { $A'.in=f(X.x)$ } A' { $A.a=A'.syn$ }

$A' \rightarrow Y$ { $A'_1.in=g(A'.in, Y.y)$ } A'_1 { $A'.syn = A'_1.syn$ }

$A' \rightarrow \varepsilon$ { $A'.syn = A'.in$ }

Evaluating attributes



Eliminating Left Recursion (in general)

- $A \rightarrow A_1Y$ $\{A.a = g(A_1.a, Y.y)\}$
- $A \rightarrow X$ $\{A.a = f(X.x)\}$

A.a:

Synthesized Attribute

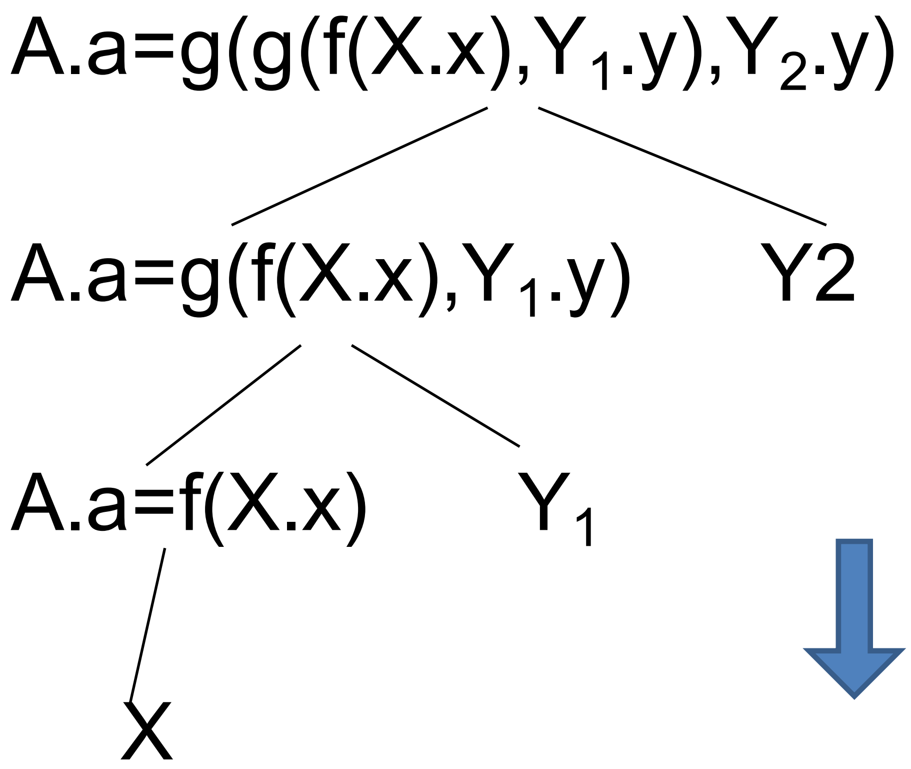
- $A \rightarrow X$ $\{R.i = f(X.x)\}$
  R $\{A.a = R.s\}$
- $R \rightarrow Y$ $\{R_1.i = g(R.i, Y.y)\}$
  R_1 $\{R.s = R_1.s\}$
- $R \rightarrow \varepsilon$ $\{R.s = R.i\}$

R.i:

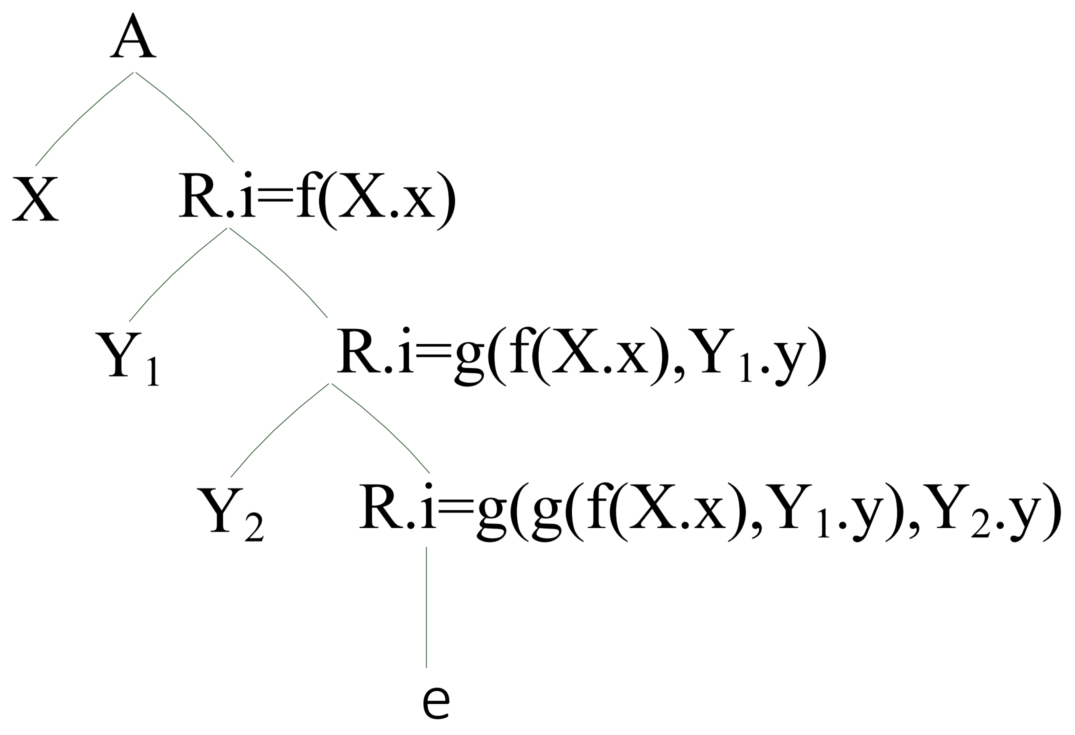
Inherited Attribute

R.s:

Synthesized Attribute



- $A \rightarrow A_1Y$ $\{A.a = g(A_1.a, Y.y)\}$
- $A \rightarrow X$ $\{A.a = f(X.x)\}$



- $A \rightarrow X$ $\{R.i = f(X.x)\}$
  R $\{A.a = R.s\}$
- $R \rightarrow Y$ $\{R_1.i = g(R.i, Y.y)\}$
  R_1 $\{R.s = R_1.s\}$
- $R \rightarrow \varepsilon$ $\{R.s = R.i\}$

Bottom-Up Translation

- We discussed already “top-down translation.”
- Works exceptionally well with L-attributed Definitions (with corresponding translation schemes)
- Nevertheless we must make sure that the underlying grammar is suitable for predictive parsing.
- **Grammar has no left-recursion and it is left-factored.**
- We also discussed “bottom up translation” for S-Directed Definitions.
- Simple Idea: Use additional stack space to store attribute values for Non-terminals. During Reduce Actions update attributes in stack accordingly.

Removing Embedding Semantic Actions

- In bottom-up evaluation scheme, the semantic actions are evaluated during the reductions.
- During the bottom-up evaluation of S-attributed definitions, we have a parallel stack to hold synthesized attributes.
- *Problem*: where are we going to hold inherited attributes?
- *A Solution*:
  - We will convert our grammar to an equivalent grammar to guarantee to the followings.
  - All embedding semantic actions in our translation scheme will be moved into the end of the production rules.
  - All inherited attributes will be copied into the synthesized attributes (most of the time synthesized attributes of new non-terminals).
  - Thus we will be evaluate all semantic actions during reductions, and we find a place to store an inherited attribute.

Removing Embedding Semantic Actions

- To transform our translation scheme into an equivalent translation scheme:
- Remove an embedding semantic action S_i, put new a non-terminal M_i instead of that semantic action.
- Put that semantic action S_i into the end of a new production rule $M_i \rightarrow \varepsilon$ for that non-terminal M_i.
- That semantic action S_i will be evaluated when this new production rule is reduced.
- The evaluation order of the semantic rules are not changed by this transformation.

Removing Embedding Semantic Actions

$A \rightarrow \{S_1\}\ X_1\ \{S_2\}\ X_2\ \ldots\ \{S_n\}\ X_n$

$\Downarrow$ remove embedding semantic actions

$A \rightarrow M_1\ X_1\ M_2\ X_2\ \ldots\ M_n\ X_n$

$M_1 \rightarrow \varepsilon\ \{S_1\}$

$M_2 \rightarrow \varepsilon\ \{S_2\}$

.

.

$M_n \rightarrow \varepsilon\ \{S_n\}$

Removing Embedding Semantic Actions

$E \to T\ R$

$R \to +\ T$ { print("+") } R_1

$R \to \varepsilon$

$T \to$ **id** { print(**id**.name) }

$\Downarrow$ remove embedding semantic actions

$E \to T\ R$

$R \to +\ T\ M\ R_1$

$R \to \varepsilon$

$T \to$ **id** { print(**id**.name) }

$M \to \varepsilon$ { print("+") }

Evaluation of Inherited Attributes

- Removing embedding actions from attribute grammar by introducing marker nonterminals

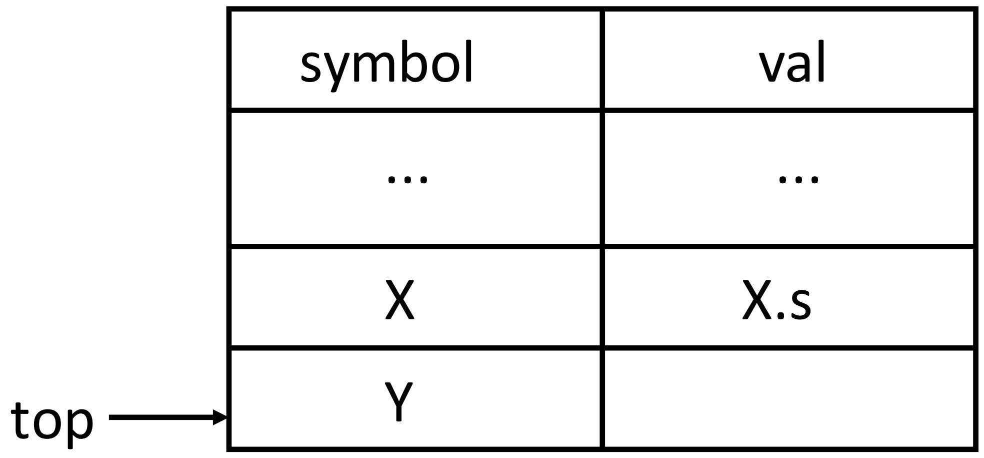
$E \rightarrow T\ R$
$R \rightarrow$ “+” T {print(‘+’)} R | “-” T {print(‘-’)} R | ε
$T \rightarrow$ **num** {print(**num**.val)}

$E \rightarrow T\ R$
$R \rightarrow$ “+” T M R | “-” T N R | ε
$T \rightarrow$ **num** {print(**num**.val)}
$M \rightarrow \varepsilon$ {print(‘+’)}
$N \rightarrow \varepsilon$ {print(‘-’)}

Evaluation of Inherited Attributes (cont.)

Inheriting synthesized attributes on the stack

$A \rightarrow X \ \{Y.i := X.s\} \ Y$



symbol	val
...	...
X	X.s
Y	

$A \rightarrow X\,M\,Y$

$M \rightarrow \varepsilon\{Y.i := X.s\}$

- If inherited attribute Y.i is denfined by the copy rule Y.i=X.s, then the value X.s can be used where Y.i is called for.
- Copy rules play an important role in the evaluation of inherited attributes.

An Example

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

$$\Downarrow$$

$D \rightarrow T \;\; L$	
$T \rightarrow$ **int**	{val[ntop] := integer;}
$T \rightarrow$ **float**	{val[ntop] := float;}
$L \rightarrow L_1$, **id**	{addtype(val[top], val[top-3]); }
$L \rightarrow$ **id**	{addtype(val[top], val[top-1]); }

An Example

Input	symbol	val	production used
int p,q,r			
p,q,r	**int**	_	
p,q,r	T	*i*	T → **int**
,q,r	T **id**	*i* *e*	
,q,r	T L	*i* _	L → **id**
q,r	T L ,	*i* _ _	
,r	T L , **id**	*i* _ _ *e*	
,r	T L	*i* _	L → L , **id**
r	T L ,	*i* _ _	
	T L , **id**	*i* _ _ *e*	
	T L	*i* _	L → L , **id**
	D		

Inheriting the value of a synthesized attribute works only if the grammar allows the position of the attribute value to be predicted.

Another Example

$S \rightarrow a\ A\ C$ {C.i := A.s}

$S \rightarrow b\ A\ B\ C$ {C.i := A.s}

$C \rightarrow c$ {C.s := g(C.i)}

$S \rightarrow a\ A\ C$ {C.i := A.s}

$S \rightarrow b\ A\ B\ M\ C$ {M.i := A.s; C.i := M.s}

$C \rightarrow c$ {C.s := g(C.i)}

$M \rightarrow \varepsilon$ {M.s := M.i}

Another Example

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

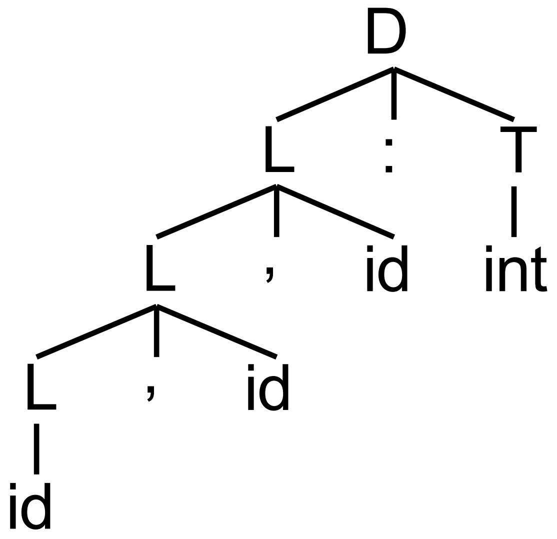
$S \rightarrow a\ A\ N\ C$ {N.i := A.s; C.i := N.s}

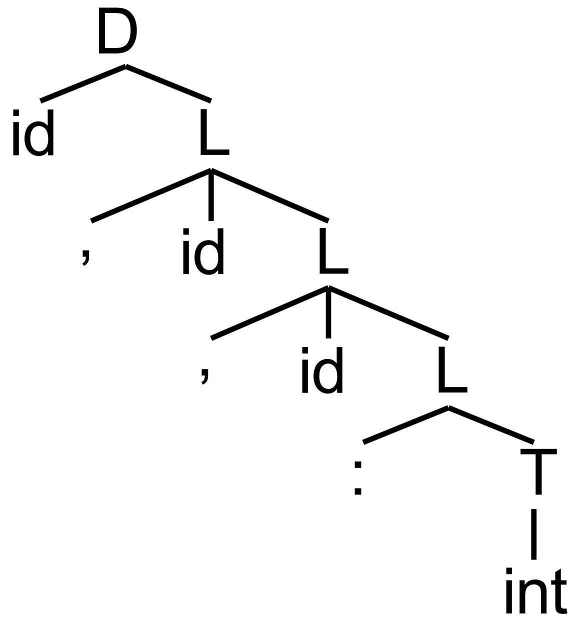
$N \rightarrow \varepsilon$ {N.s := f(N.i)}

From Inherited to Synthesized

D → L : T
L → L , **id** | **id**
T → **integer** | **char**

D → **id** L
L → , **id** L | : T
T → **integer** | **char**





Problems

- Some L-attributed definitions based on LR grammars cannot be evaluated during bottom-up parsing.

$S \rightarrow \{ L.i=0 \}\ L$

$L \rightarrow \{ L_1.i=L.i+1 \}\ L_1\ 1$

$L \rightarrow \varepsilon\ \{ print(L.i) \}$

➔ this translations scheme cannot be implemented during the bottom-up parsing

$S \rightarrow M_1\ L$

$L \rightarrow M_2\ L_1\ 1$

$L \rightarrow \varepsilon \quad \{ print(s[top]) \}$

$M_1 \rightarrow \varepsilon \quad \{ s[ntop]=0 \}$

$M_2 \rightarrow \varepsilon\ \{ s[ntop]=s[top]+1 \}$

➔ But since $L \rightarrow \varepsilon$ will be reduced first by the bottom-up parser, the translator cannot know the number of 1s.

Problems

The modified grammar cannot be LR grammar anymore.

$L \rightarrow L\ b$ ➔ $L \rightarrow M\ L\ b$

$L \rightarrow a$ ➔ $L \rightarrow a$ NOT LR-grammar

$M \rightarrow \varepsilon$

$S' \rightarrow .L, \$$

$L \rightarrow .\ M\ L\ b, \$$

$L \rightarrow .\ a, \$$

$M \rightarrow .,a$ ➔ shift/reduce conflict

Summary

- **Semantic analysis** verifies that a syntactically valid program is correctly-formed and computes additional information about the meaning of the program.
- Syntax-directed definitions
- Inherited and Synthesized Attributes
- Attribute Grammars
- Dependency Graphs
- **S-Attributed Definitions**
- **L-Attributed Definitions**

Reference

- Compilers Principles, Techniques & Tools (Second Edition) Alfred Aho., Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007
- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu