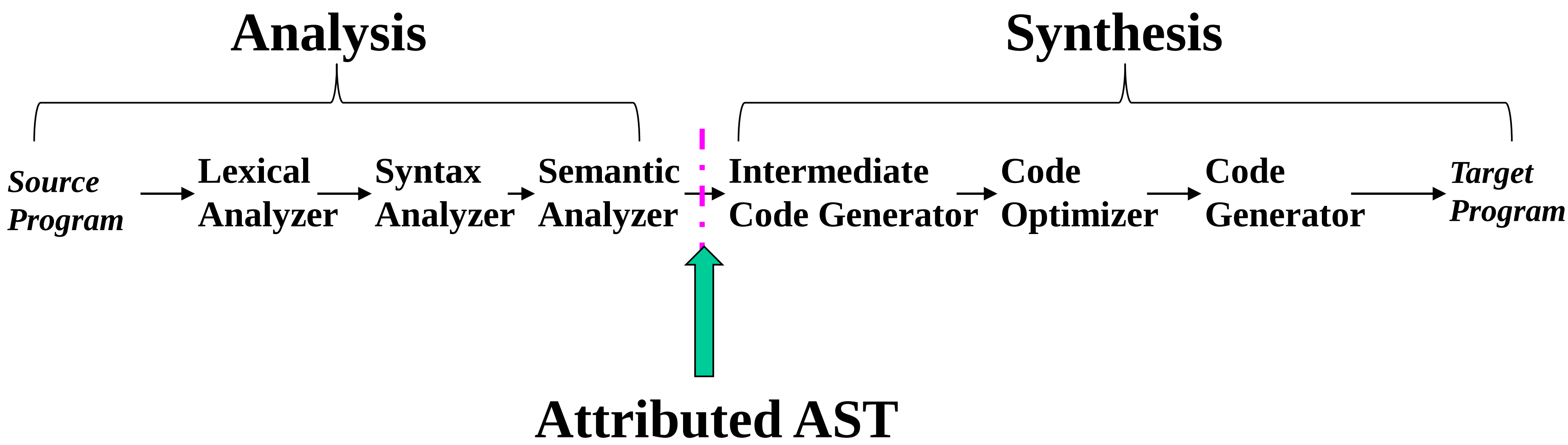
Syntax-Directed Translation



Syntax-Directed Translation

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
  - Attributes for expressions:
    - type of value: int, float, double, char, string,…
    - type of construct: variable, constant, operations, …
  - Attributes for variables: name, type
- 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.**

E:= E+T | T
T:= T*F | F
F:= digit | (E)

Syntax-Directed Definitions and Translation Schemes

- When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes/ Syntax-Directed Translation**
- **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

- A syntax-directed definition （SDD）
  - 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.

    Specify how to compute attribute values of symbols

- Production: $A \rightarrow \alpha$ Semantic rule $b=f(c_1,c_2,\ldots,c_n)$
  - b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in $A \rightarrow \alpha$.
  - b is an inherited attribute of one of the grammar symbols in α and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in $A \rightarrow \alpha$.

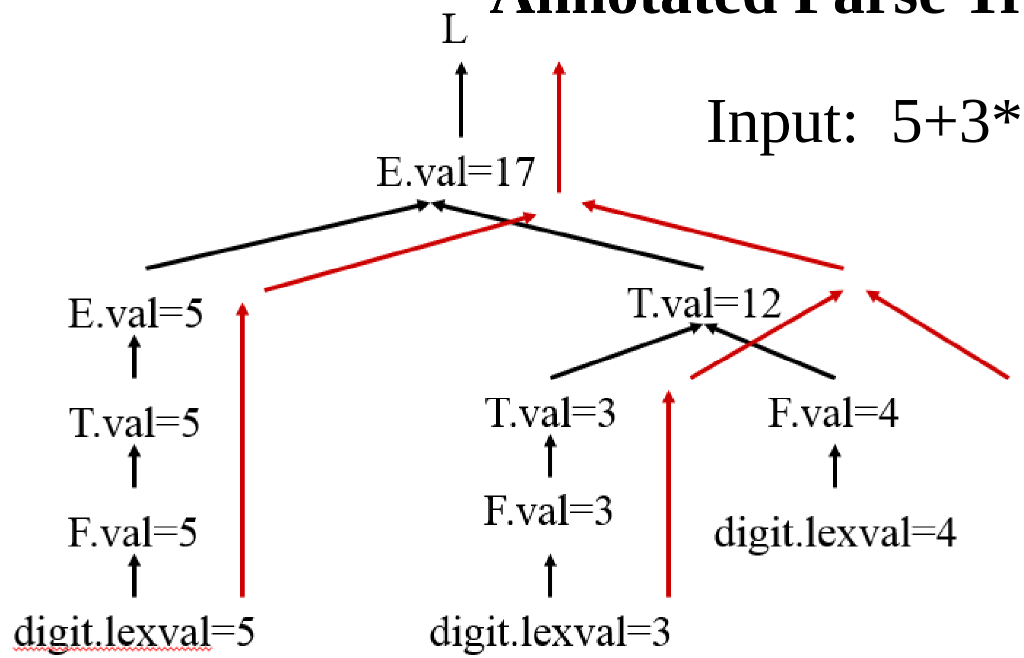
Synthesized Attributes

- Production: $A \rightarrow \alpha$ Semantic rule $b=f(c_1,c_2,\ldots,c_n)$
  - b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in $A \rightarrow \alpha$.

Production	**Semantic Rules**
$L \rightarrow E$	**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**

Annotated Parse Tree

Input: 5+3*4



- **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).**

Inherited Attributes

Production: $A \rightarrow \alpha$ **Semantic rule** $b=f(c_1,c_2,\ldots,c_n)$

- ***b*** **is an inherited attribute of one of the grammar symbols in α and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in $A \rightarrow \alpha$.**

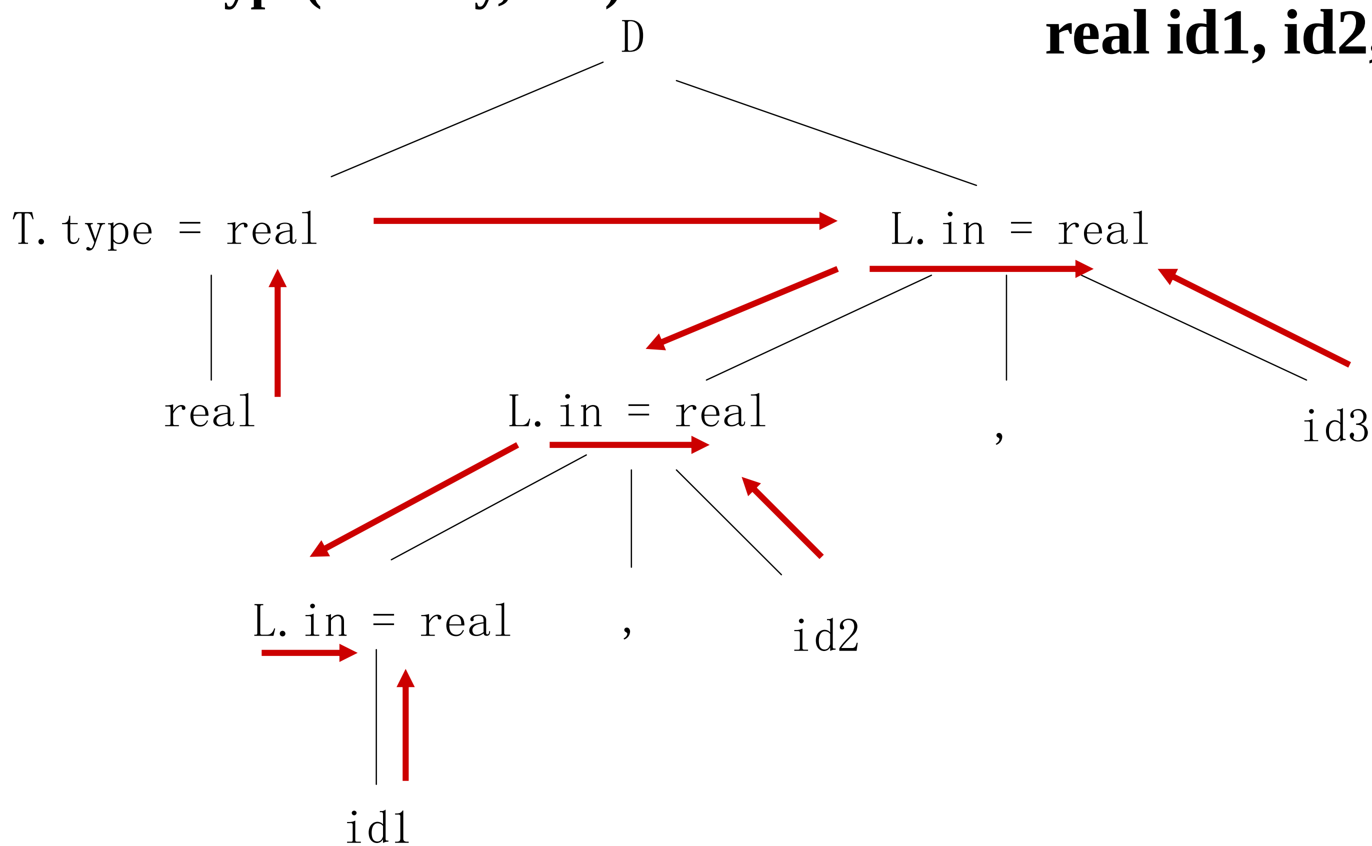
Production	Semantic Rules
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = real
$L \rightarrow L_1$, **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
$L \rightarrow$ **id**	addtype(**id**.entry,L.in)

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

Inherited Attributes (cont.)

Production	Semantic Rules
$D \rightarrow T\ L$	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)

real id1, id2, id3



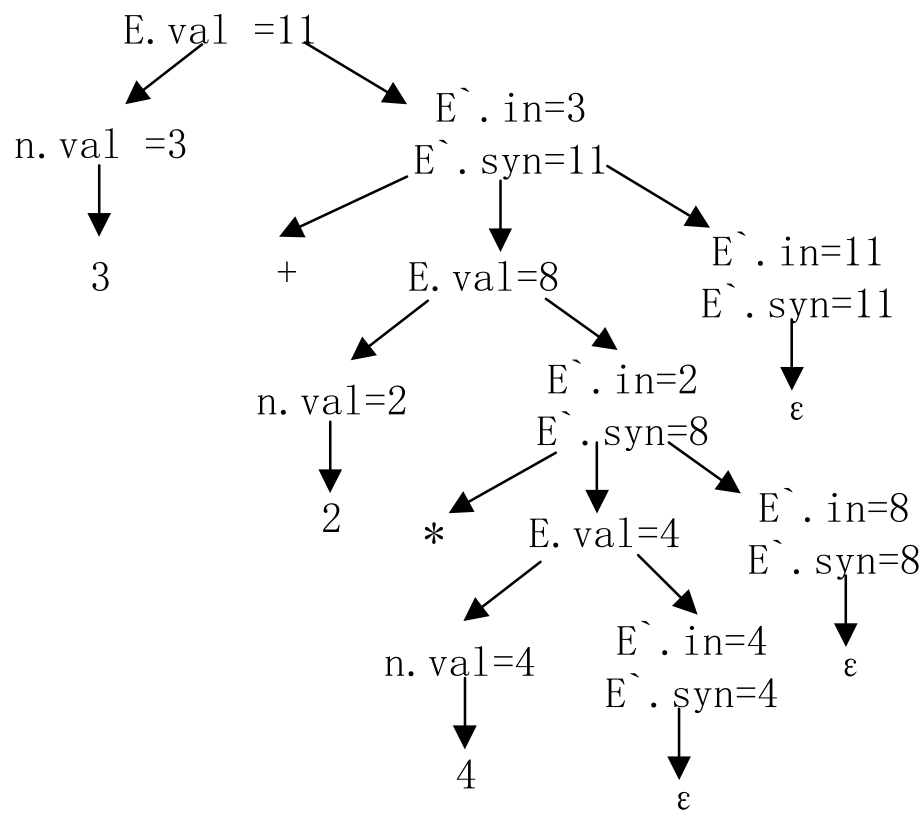
Synthesized and Inherited Attributes

- Sometimes both synthesized and inherited attributes are required to evaluate necessary information

E:= n E’
E’:= +EE’ | *EE’ | ε

3+2*4

Production	Semantic rule
E:= n E’	E’.in= n.val; E.val = E’.syn
E’:= + EE’$_1$	E’$_1$.in = E’.in + E.val; E’.syn = E’$_1$.syn
E’ :=* EE’$_1$	E’$_1$.in = E’.in * E.val; E’.syn = E’$_1$.syn
E’: = ε	E’.syn = E’.in



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 $c_1, c_2, \ldots, c_k$

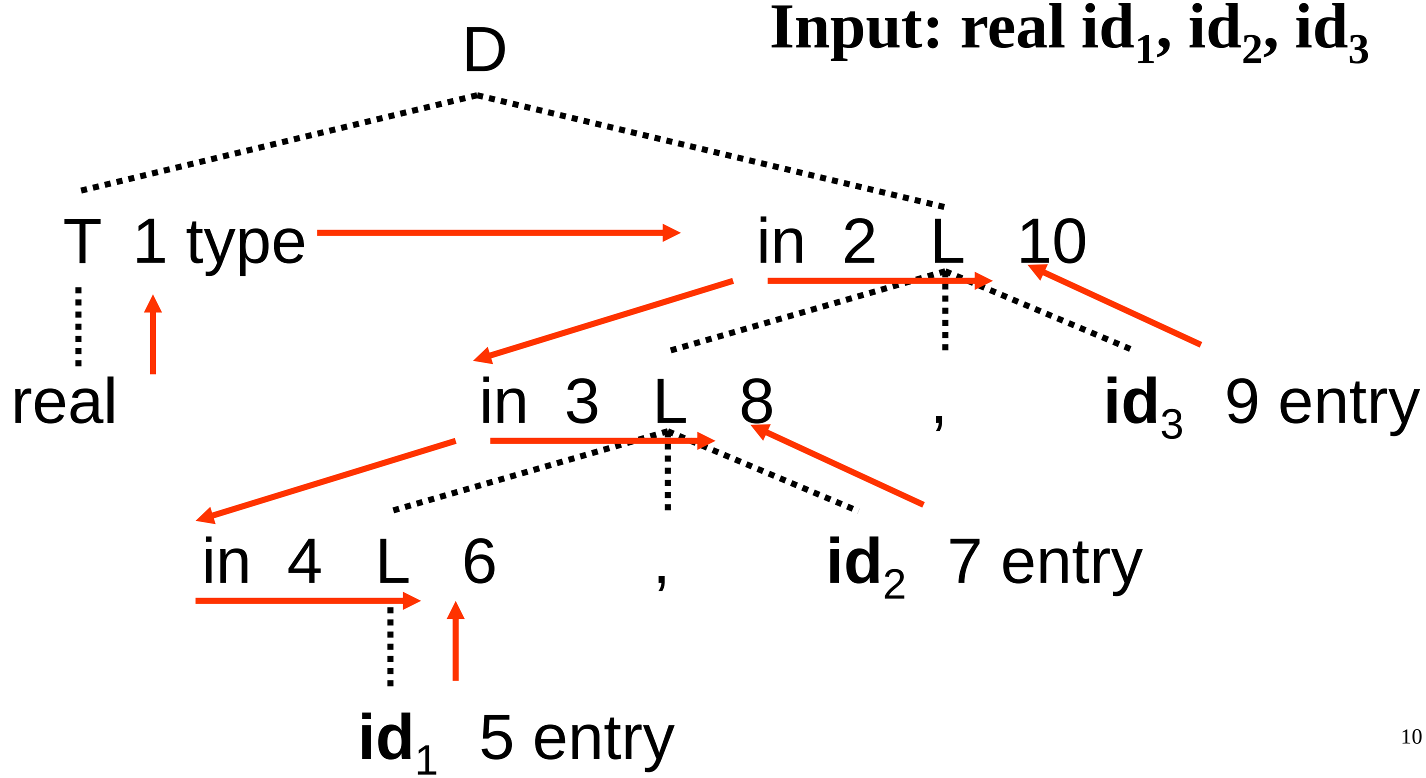
- The dependencies of attributes can be represented by a directed graph called dependency graph

Dependency Graphs

Production	Semantic Rules
$D \rightarrow T\ L$	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)

Input: real id_1, id_2, id_3

Evolution
Order is a topological order of the dependency graph
e.g:
1→2→3…

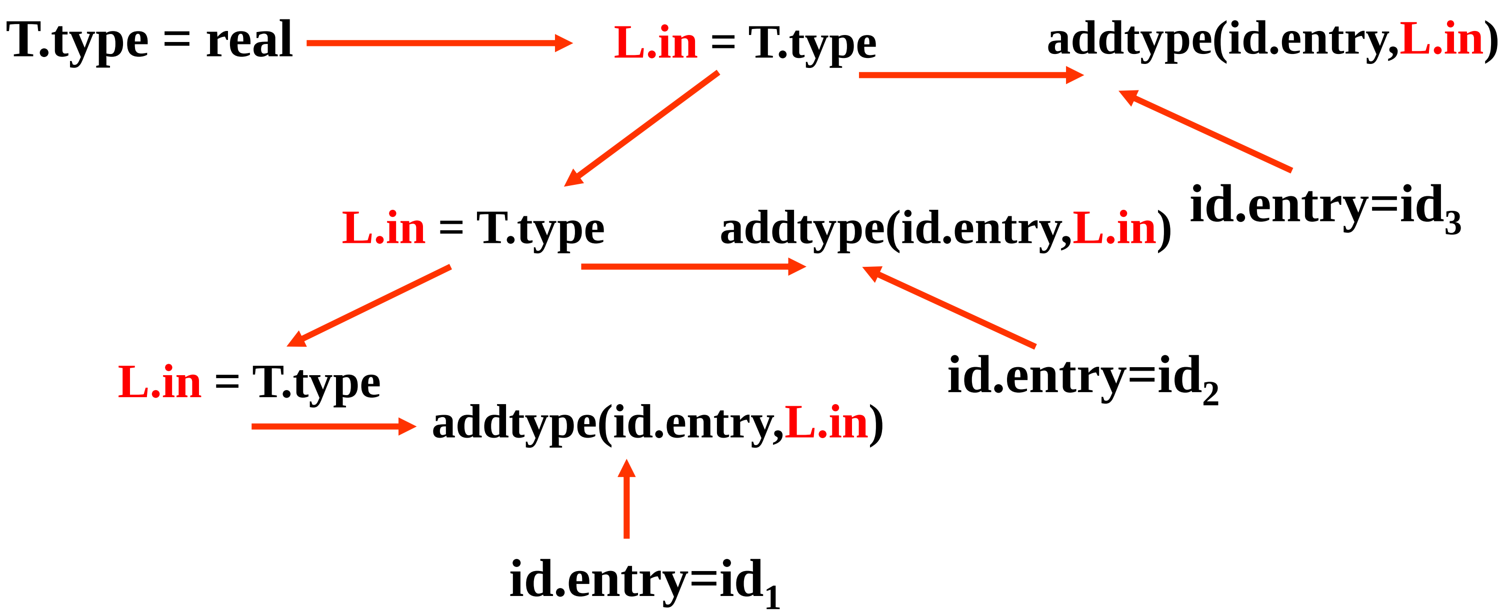


Dependency Graphs

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

Input: real id_1, id_2, id_3

Evaluation



Dependency Graphs – quiz

Production	Semantic Rules
$D \rightarrow T\ L$	$L.in$ = $T.type$
$T \rightarrow$ 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,$L.in$)

Input: int p, g

Draw dependency graph and one of its evaluation order

Evaluation of semantic rules

- Parse-tree methods (general approach for any acyclic dependency graphs)
  1. Build a parse tree for each input
  2. Build a dependency graph from the parse tree
  3. Obtain evaluation order from a topological order of the dependency graph
- Rule-based methods (parsing time)
  1. Predetermine the order of attribute evaluation for each production, e.g., translation schemes
- Oblivious methods
  1. Evaluation order is independent of semantic rules
  2. Evaluation order forced by parsing methods
  3. Restrictive in acceptable attribute definitions
     - E.g.: S-Attributed Definitions and L-Attributed Definitions

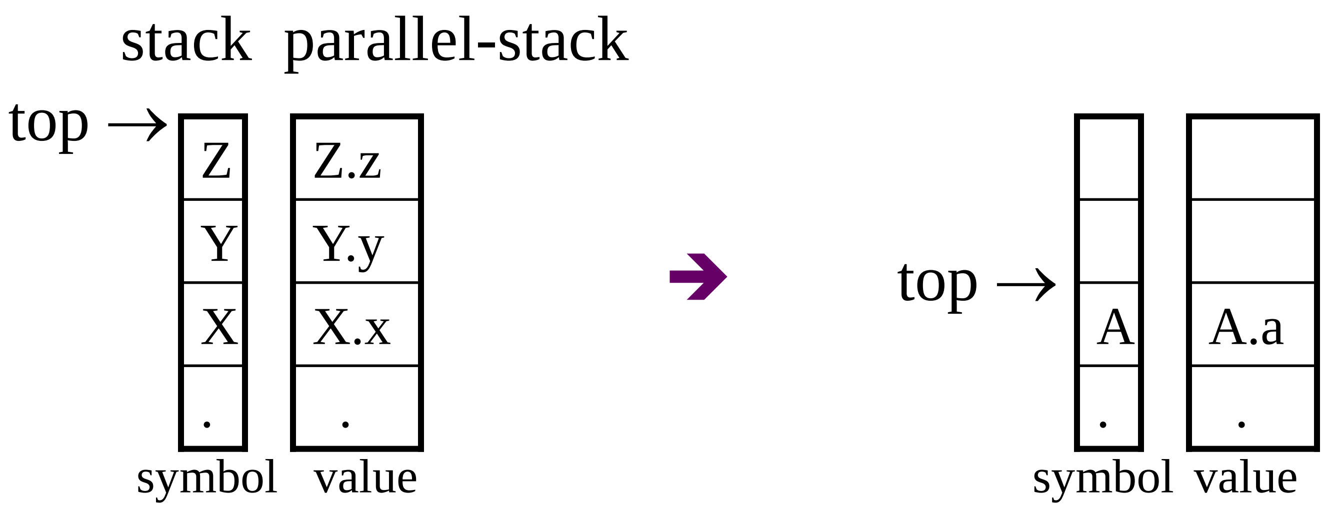
S-Attributed Definitions
L-Attributed Definitions

- 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.
- Implementation of S-Attributed Definitions and L-Attributed Definitions are easy (we can evaluate semantic rules in a single pass during the parsing).
- Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions.

Bottom-Up Evaluation of S-Attributed Definitions

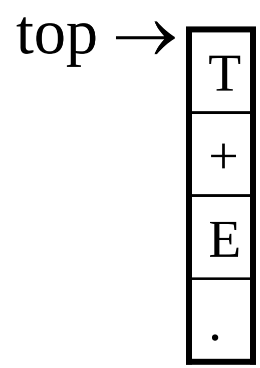
- S-attributed Definition: Syntax-Directed Definition using only Synthesized attributes.
- Stack of a LR parser contains states.
- Recall that each state corresponds to some grammar symbol (and many different states might correspond to the same grammar symbol)
- Keep attribute values of grammar symbols in stack
- Evaluate attribute values at each reduction

$A \rightarrow XYZ$ $\quad$ A.a=f(X.x,Y.y,Z.z) $\quad$ where all attributes are synthesized.



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

Production	Semantic Rules
$L \rightarrow E$	print(val[top])
$E \rightarrow E1 + T$	val[top] = val[top-2] + val[top]
$E \rightarrow T$	
$T \rightarrow T1 * F$	val[top] = val[top-2] * val[top]
$T \rightarrow F$	
$F \rightarrow (E)$	val[top] = val[top-1]
$F \rightarrow$ **digit**	

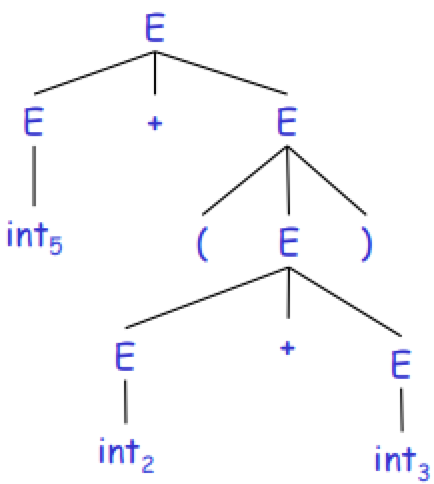


Decrement top before assignment

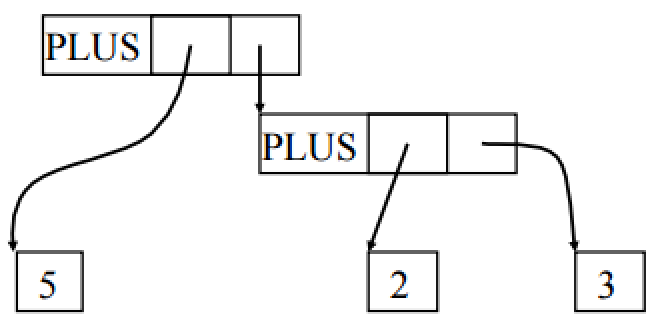
- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Create Abstract Syntax Tree

Production	**Semantic Rules**
$L \rightarrow E$	L.exp =E.exp
$E \rightarrow E1 + T$	E.exp = Exp(Plus, E1.exp , T.exp)
$E \rightarrow T$	E.exp = T.exp
$T \rightarrow T1 * F$	T.exp = Exp(Mul,T1.exp, F.exp)
$T \rightarrow F$	T.exp =F.exp
$F \rightarrow (E)$	F.exp =E.exp
$F \rightarrow$ **digit**	F.exp =digit.val



5+(2+3)



Constructing SLR(1) Parsing Table)

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$

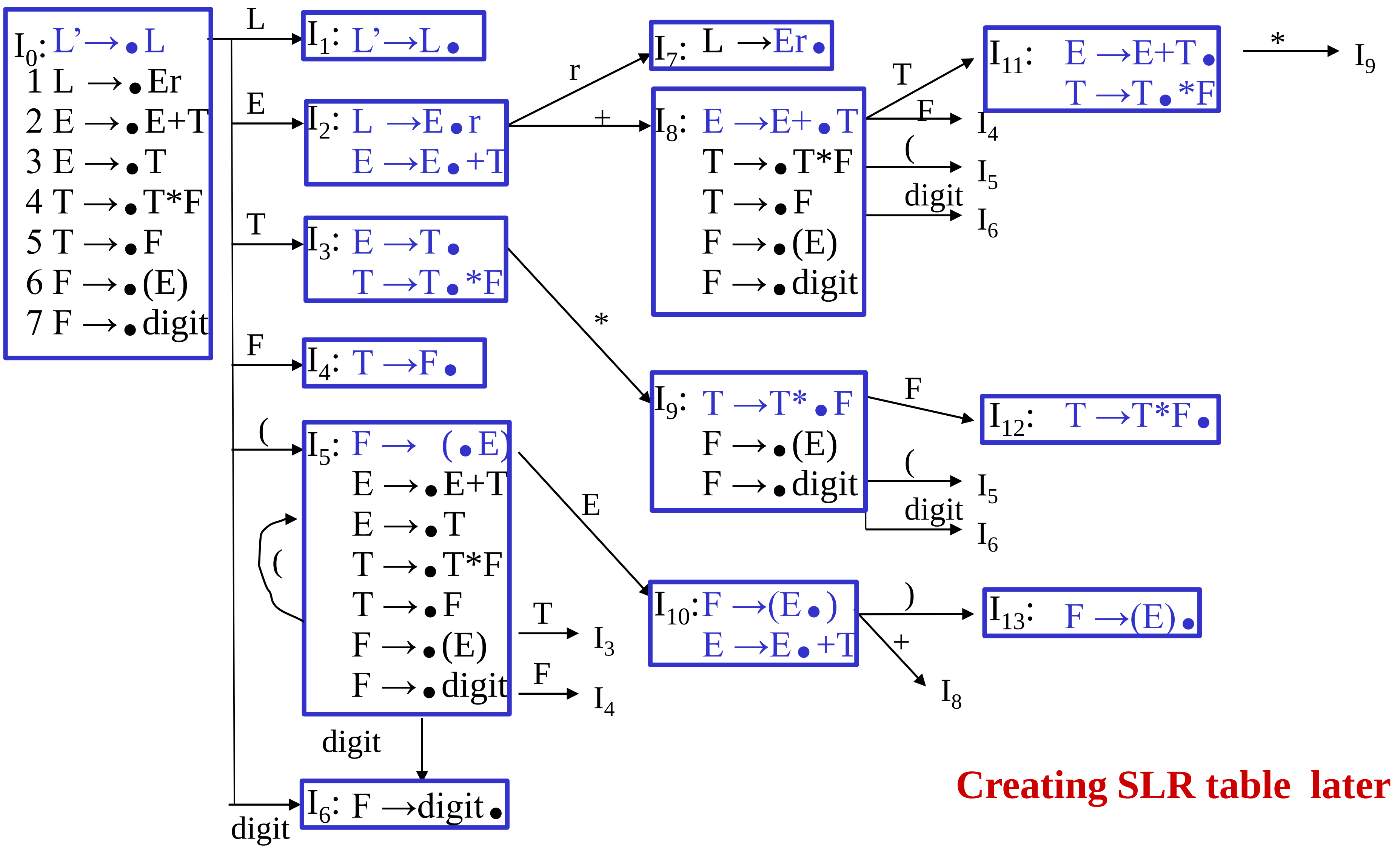
2. Create the parsing action table as follows:
   - If a is a terminal, $A\rightarrow\alpha.\mathbf{a}\beta$ in I_i and goto(I_i,**a**)=I_j then action[i,a] is ***shift j*.**
   - If $A\rightarrow\alpha.$ is in I_i, then action[i,a] is ***reduce A→α*** **for all a in FOLLOW(A) where A≠S'.**
   - If $S'\rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table :
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S'\rightarrow .S$

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	

The extension is same for LR(1) and LALR(1)

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

  $A \rightarrow X_1\ X_2\ \ldots\ X_{j-1}\ X_j\ \ldots\ X_n$ $\quad b=f(c_1,c_2,\ldots,c_n)$

- b 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, …, 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)

S-Attributed Attribute Grammar?

L-Attributed Attribute Grammar ?

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)

Why?

- 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:* $A \rightarrow \{ \ldots \}\ X\ \{ \ldots \}\ Y\ \{ \ldots \}$

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
$T \longrightarrow T_1 * F$	$T.val := T_1.val * F.val$

Yield the following production and semantic action:

$T \longrightarrow T_1 * F$ $\{T.val := T_1.val * F.val\}$

Designing 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

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

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

This will not work…

Why?

On the other hand this is good:

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

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

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…

Why?

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 our 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

A Translation Scheme Example

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

$E \rightarrow T\ R$

$R \rightarrow +\ T\ \{\ print(\text{"+"})\ \}\ R1$

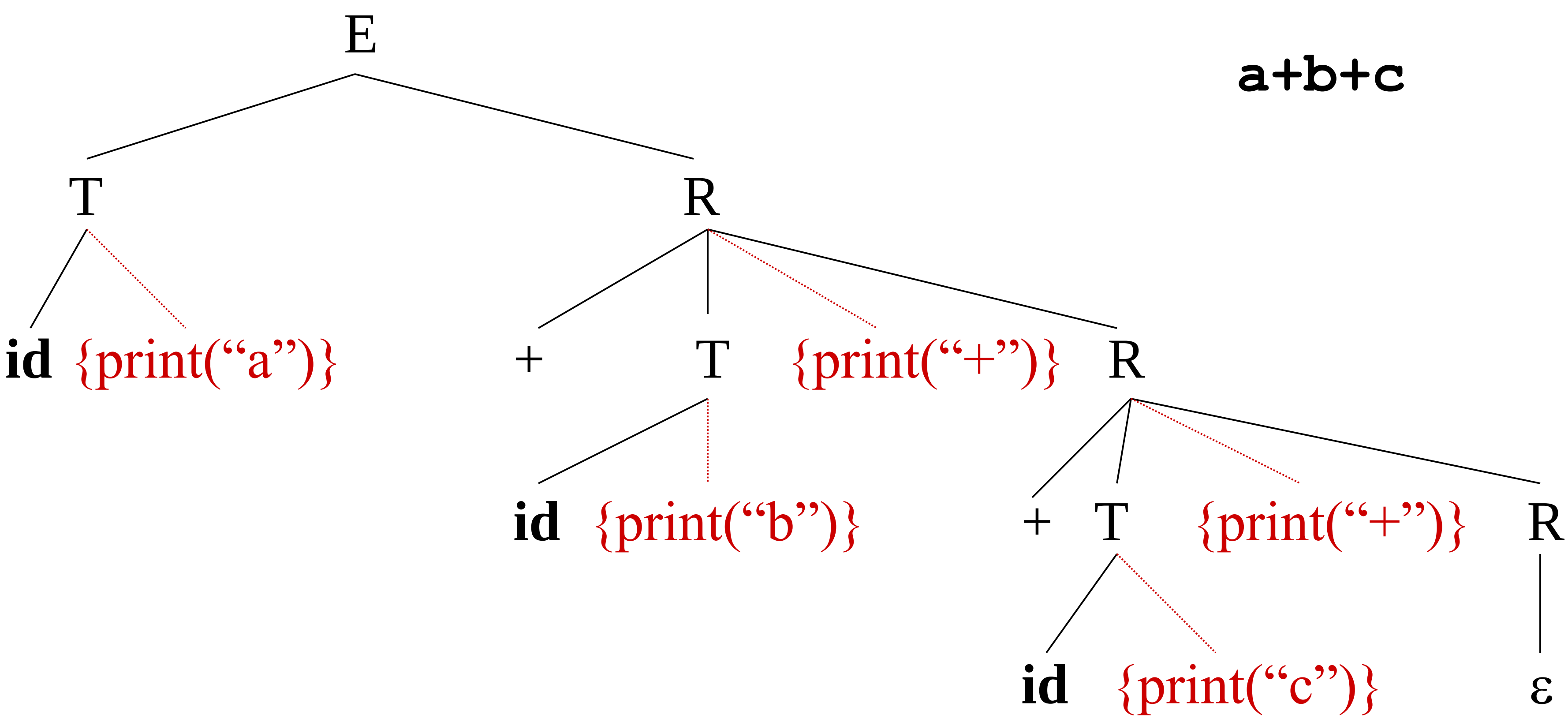
$R \rightarrow \varepsilon$

$T \rightarrow \textbf{id}\ \{\ print(\textbf{id}.name)\ \}$

a+b+c ➔ ab+c+

infix expression   postfix expression

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

$E \rightarrow T\ R$
$R \rightarrow +\ T$ { print("+") } $R1$
$R \rightarrow \varepsilon$
$T \rightarrow$ **id** { print(**id**.name) }

Left-most derivation: **`a+b+c`**

- $E \rightarrow T\ R$
- → **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?

Top-Down Translation

- We will look at the implementation of L-attributed definitions during predictive parsing.
- Instead of the syntax-directed definitions, 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.

A Translation Scheme with Inherited Attributes

- For each nonterminal A, construct a function ParseA(in){return s;}
  - inherited attributes in $\rightarrow$ formal parameters
  - synthesized attributes s $\rightarrow$ returned values

- The code associated with each production does the following
  - for each terminal X with synthesized attribute x,
    save X.x; match(X);
  - for nonterminal B, c := B(b_1, b_2, …, b_k); where b_1, b_2, …, b_k are the values for the attributes and c is a variable to store s-attributes of B
  - for each semantic action, copy the action to the parser, replacing references to attributes by the corresponding variables

Predictive Parsing (of Inherited Attributes)

D → T { 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

```
procedure parseD() {
    Type t= parseT();
    Type Lin =t
    parseL(Lin);
}
procedure parseT() {
    Type t;
    if (currtoken is int) { currtoken++; t= TYPEINT; }
    else if (currtoken is real) { currtoken++; t= TYPEREAL; }
    else { error("unexpected type"); }
    return t;
}
procedure parseL( Type Lin ) {
    if (currtoken is id) { SymEntry entry=getEnrty(ST, currtoken);
                           addtype( entry,  Lin ); parseL(Lin); }
    else if (currtoken is endmarker)  { }
    else { error("unexpected token"); }
}
```

a synthesized attribute (an return value)

an inherited attribute (an input parameter)

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.)

inherited attribute   synthesized attribute

$E \rightarrow T$ { 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 }

$E \rightarrow E_1 + T$ { 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 }

Eliminating Left Recursion (in general)

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

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

a left recursive grammar with synthesized attributes (a,y,x).

$\Downarrow$ eliminate left recursion

inherited attribute of the new non-terminal

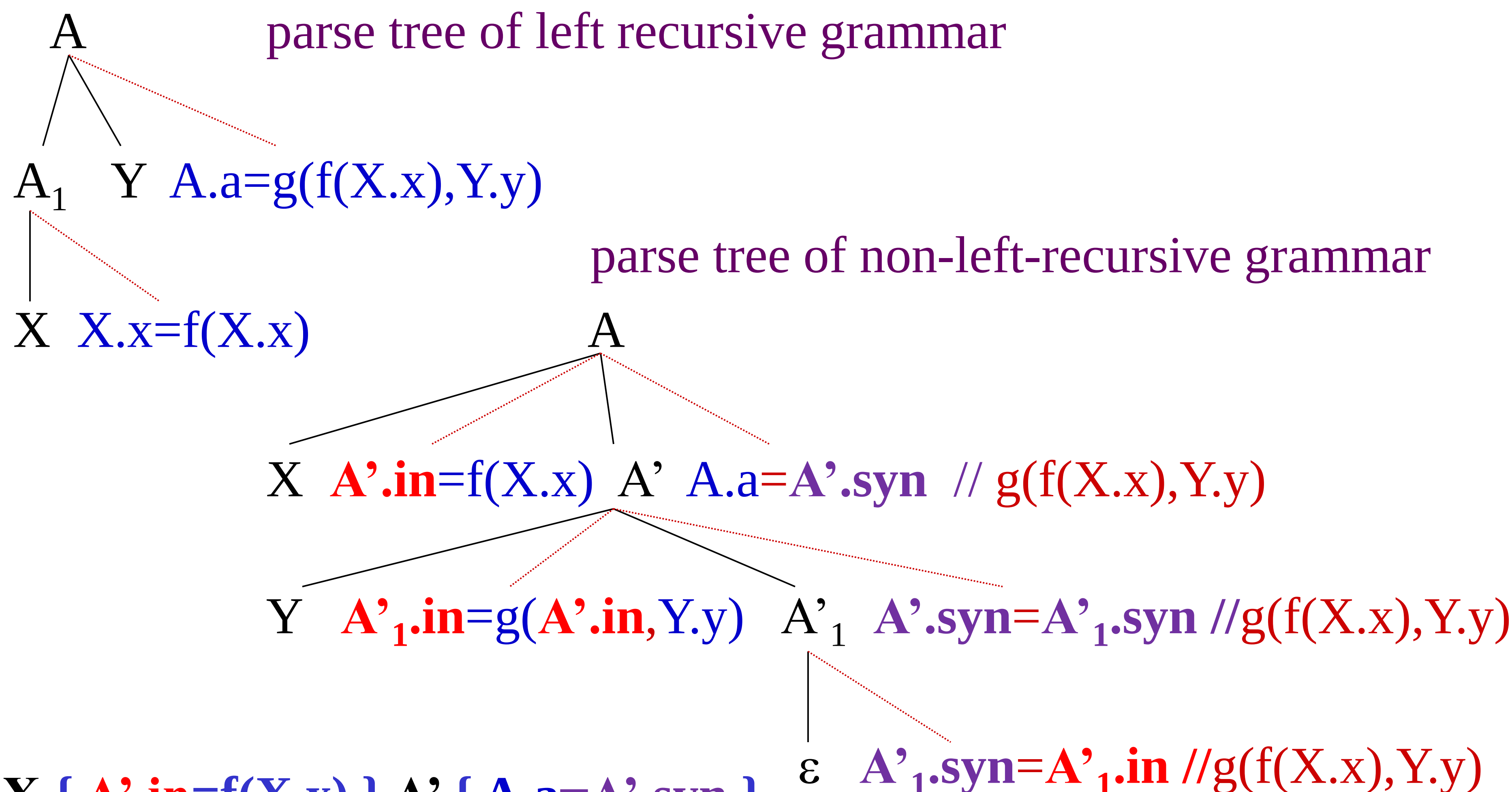
synthesized attribute of the new non-terminal

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

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

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

Evaluating attributes



A → X { A'.in=f(X.x) } A' { A.a=A'.syn }

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

A' → ε { A'.syn = A'.in }

Mid-Milestone

- S-attributed Definitions => Bottom-up parsing
- L-attributed Definitions => Top-down parsing
- S-attributed Definitions are also L-attributed Definitions
- S-attributed Definitions => Top-down parsing
- 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.**

Bottom-Up Translation

- **How to do bottom-up translation for L-Directed Definitions?**

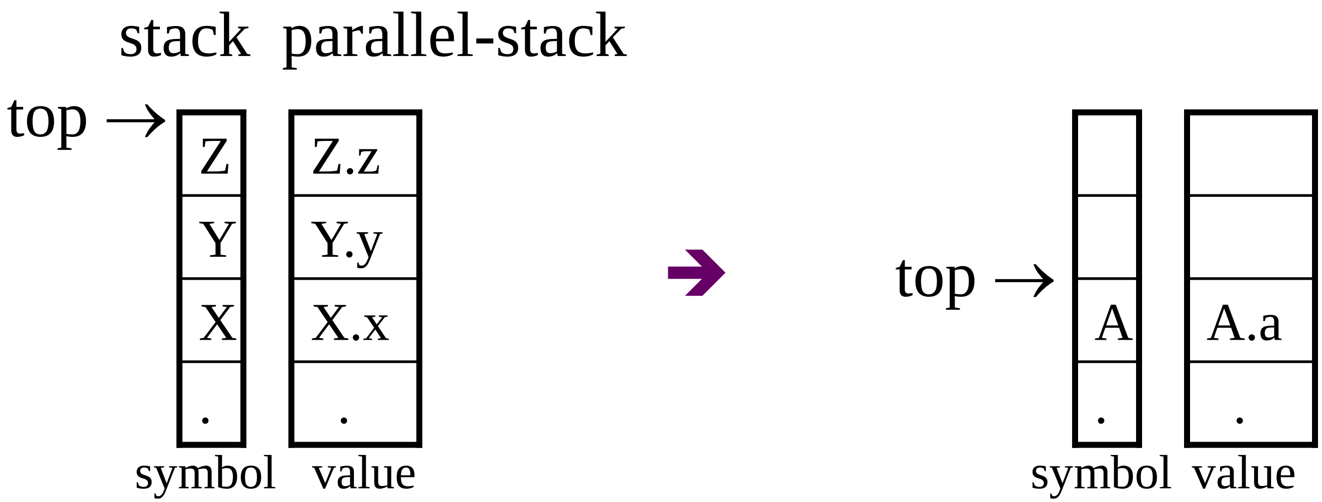
Simple Idea:

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?**

Removing Embedding Semantic Actions

- ***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 evaluate all semantic actions during reductions, and we find a place to store an inherited attribute.

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)}

$\Downarrow$ **remove embedding semantic actions**

$E \rightarrow T\ R$

$R \rightarrow$ “+” T M R | “-” T N R | ε

$T \rightarrow$ **num** {print(**num**.val)}

$M \rightarrow \varepsilon$ {print(‘+’)}

$N \rightarrow \varepsilon$ {print(‘-’)}

Removing Embedding Semantic Actions

- To transform our translation scheme into an equivalent translation scheme:

1. Remove an embedding semantic action S_i, put new a non-terminal M_i instead of that semantic action.
2. Put that semantic action S_i into the end of a new production rule $M_i \rightarrow \varepsilon$ for that non-terminal M_i.
3. That semantic action S_i will be evaluated when this new production rule is reduced.
4. **The evaluation order of the semantic rules are not changed by this transformation.**

Evaluation of Inherited Attributes (cont.)

- Inheriting synthesized attributes on the stack

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

$A \rightarrow X \ M \ Y$

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

top

Y	
X	X.s
…	…
symbol	Val

- If inherited attribute Y.i is defined 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 := int;}
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);}

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

Problems

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

$S \rightarrow \{ L.i=0 \}\ L$ ➔ this translations scheme cannot be implemented

$L \rightarrow \{ L_1.i=L.i+1 \}\ L_1\ 1$ during the bottom-up parsing

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

S➔L➔L1➔L11➔L111➔L1111➔1111

0 1 2 3 4 print(4)

$S \rightarrow M_1\ L$

$L \rightarrow M_2\ L_1\ 1$ ➔ But since $L \rightarrow \varepsilon$ will be reduced first by the bottom-up

$L \rightarrow \varepsilon\quad \{ print(s[top]) \}$ parser, the translator cannot know the number of 1s.

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

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

Problems

- The modified grammar cannot be LR grammar anymore.

$L \rightarrow \{action\}\ L\ b$		$L \rightarrow M\ L\ b$	
$L \rightarrow a$	➔	$L \rightarrow a$	NOT LR-grammar
		$M \rightarrow \varepsilon$	

$S' \rightarrow \bullet L, \$$

$L \rightarrow \bullet\ M\ L\ b, \$$

$L \rightarrow \bullet\ a, \$$

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

Bottom-up translation in Yacc/Bison

```
declaration: class type namelist ;

class: GLOBAL { $$ = G; }
     | LOCAL { $$ = L; }
     ;

type: REAL { $$ = R; }
    | INTEGER { $$ = I; }
    ;
namelist: NAME { mksym($0, $-1, $1); }
```

Stack: G L name
$-1 $0 $1

Yacc/Bison symbol values can act as *inherited attributes* or *synthesized attributes*.

Compute the value of a sequence of digits

1、Write attribute grammar：

$$Number \rightarrow Number_1 \ \ Digit$$

$$Number \rightarrow Digit$$

$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid$$

2、 Write attribute grammar

$$Number \rightarrow Digit \ \ Number_1$$

$$Number \rightarrow Digit$$

$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid$$