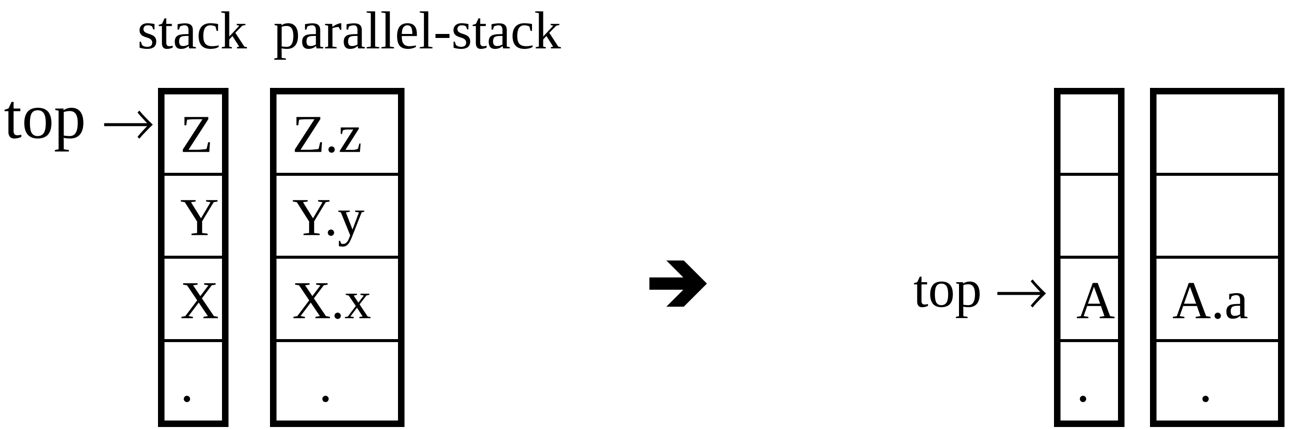
S-Attributed Definitions

- 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).
- Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

Bottom-Up Evaluation of S-Attributed Definitions

- We put the values of the synthesized attributes of the grammar symbols into a parallel stack.
  - When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.
- We evaluate the values of the attributes during reductions.

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

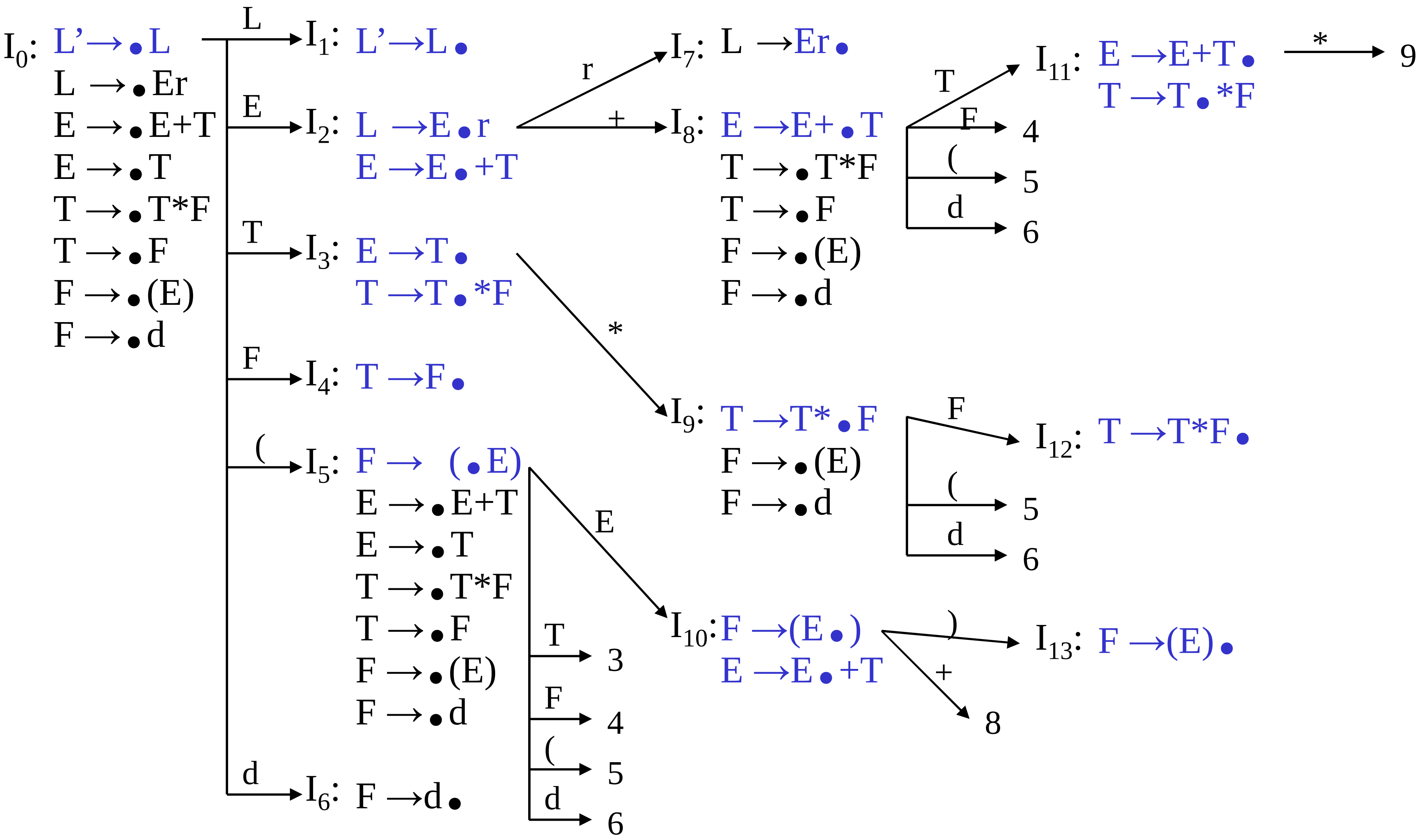


Bottom-Up Eval. of S-Attributed Definitions (cont.)

Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(val[top-1])
$E \rightarrow E_1 + T$	val[ntop] = val[top-2] + val[top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val[ntop] = val[top-2] * val[top]
$T \rightarrow F$	
$F \rightarrow (E)$	val[ntop] = val[top-1]
$F \rightarrow$ **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



Bottom-Up Evaluation -- Example

- 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	d.lexval(5) into val-stack
0d6	5	+3*4r	F→d	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	d.lexval(3) into val-stack
0E2+8d6	5-3	*4r	F→d	F.val=d.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	d.lexval(4) into val-stack
0E2+8T11*9d6	5-3-4	r	F→d	F.val=d.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	

Top-Down Evaluation (of S-Attributed Definitions)

Productions	**Semantic Rules**
$A \rightarrow B$	print(B.n0), print(B.n1)
$B \rightarrow \mathbf{0}\ B_1$	$B.n0=B_1.n0+1$, $B.n1=B_1.n1$
$B \rightarrow \mathbf{1}\ B_1$	$B.n0=B_1.n0$, $B.n1=B_1.n1+1$
$B \rightarrow \varepsilon$	$B.n0=0$, $B.n1=0$

where B has two synthesized attributes (n0 and n1).

Top-Down Evaluation (of S-Attributed Definitions)

- Remember that: In a recursive predicate parser, each non-terminal corresponds to a procedure.

```
procedure A() {
    call B();                                                   A → B
}
procedure B() {
    if (currtoken=0)  { consume 0; call B(); }                  B → 0 B
    else if (currtoken=1) { consume 1; call B(); }              B → 1 B
    else if (currtoken=$)  {}    // $ is end-marker             B → ε
    else error("unexpected token");
}
```

Top-Down Evaluation (of S-Attributed Definitions)

```
procedure A() {
    int n0,n1;
    call B(&n0,&n1);
    print(n0);   print(n1);
}
procedure B(int *n0, int *n1) {
    if (currtoken=0)
        {int a,b; consume 0; call B(&a,&b); *n0=a+1; *n1=b;}
    else if (currtoken=1)
        { int a,b; consume 1; call B(&a,&b); *n0=a; *n1=b+1; }
    else if (currtoken=$) {*n0=0; *n1=0; }    // $ is end-marker
    else error("unexpected token");
}
```

Synthesized attributes of non-terminal B are the output parameters of procedure B.

All the semantic rules can be evaluated at the end of parsing of production rules

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 Definitions

- A syntax-directed definition is **L-attributed** if each inherited attribute of X_j, where $1 \leq j \leq n$, on the right side of $A \rightarrow X_1X_2...X_n$ depends only on:
  1. The attributes of the symbols $X_1,...,X_{j-1}$ to the left of X_j in the production and
  2. the inherited attribute of A

- Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

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.