Syntax-Directed Translation

Part IV

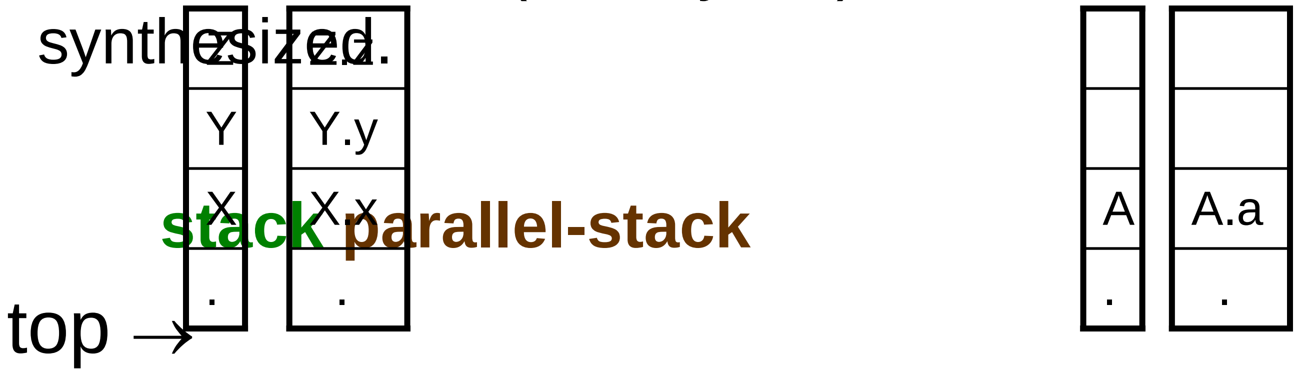
Implementation of S-Attributed Definitions

- Syntax-directed definitions are used to specify syntax-directed translations.
- 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).
- To implement S-Attributed Definitions and L-Attributed Definitions are easy (we can evaluate semantic rules in a single pass during the parsing).
- To create a translator for an arbitrary syntax-directed definition can be difficult.
- 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.

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

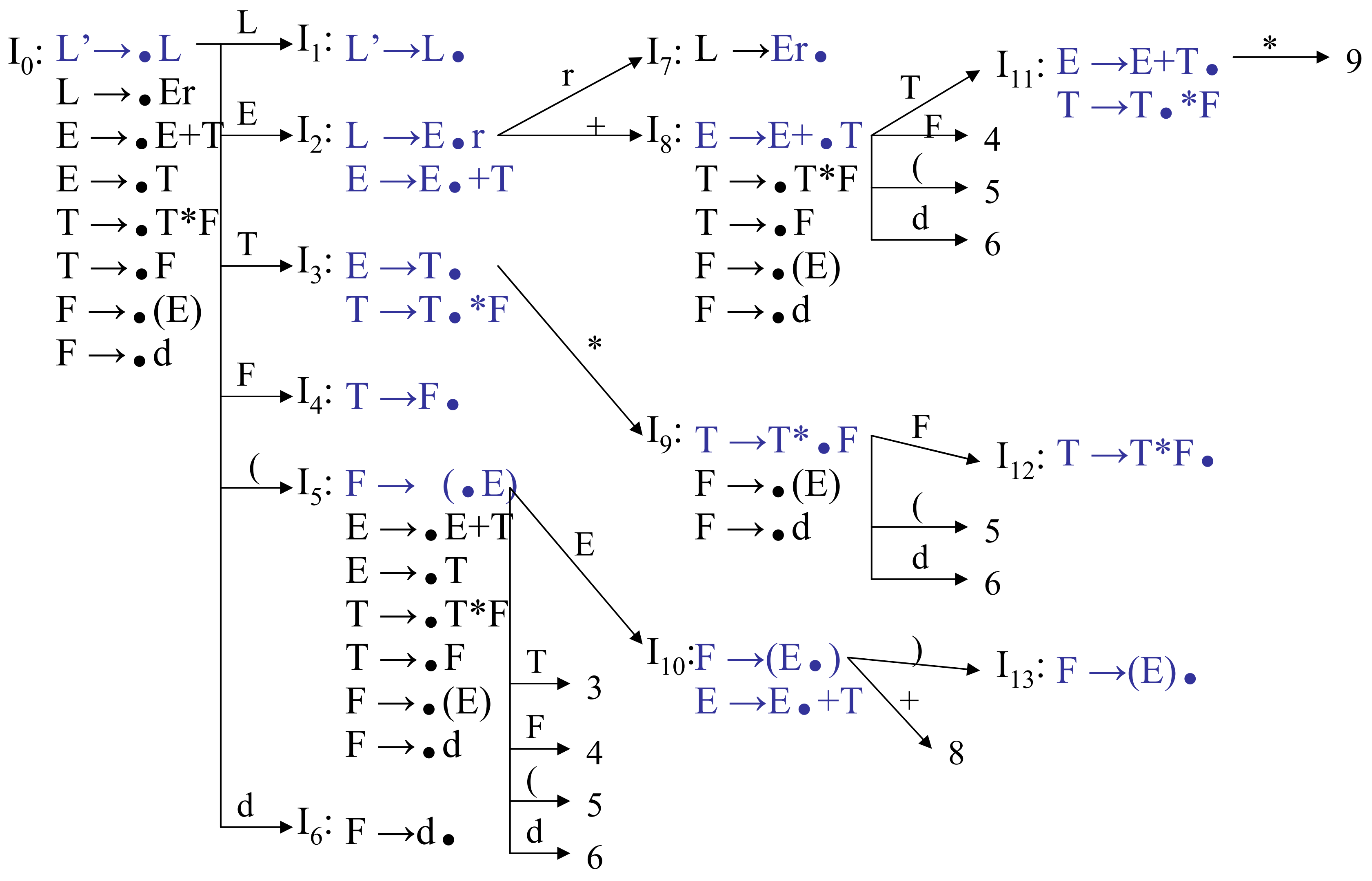


Bottom-Up Eval. of S-Attributed Definitions (cont.)

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

Implementation of 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.

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

- 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 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

- **Postfix SDT**: SDT with all actions at the right end of the production bodies
  - Implementation?

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("+") } R_1

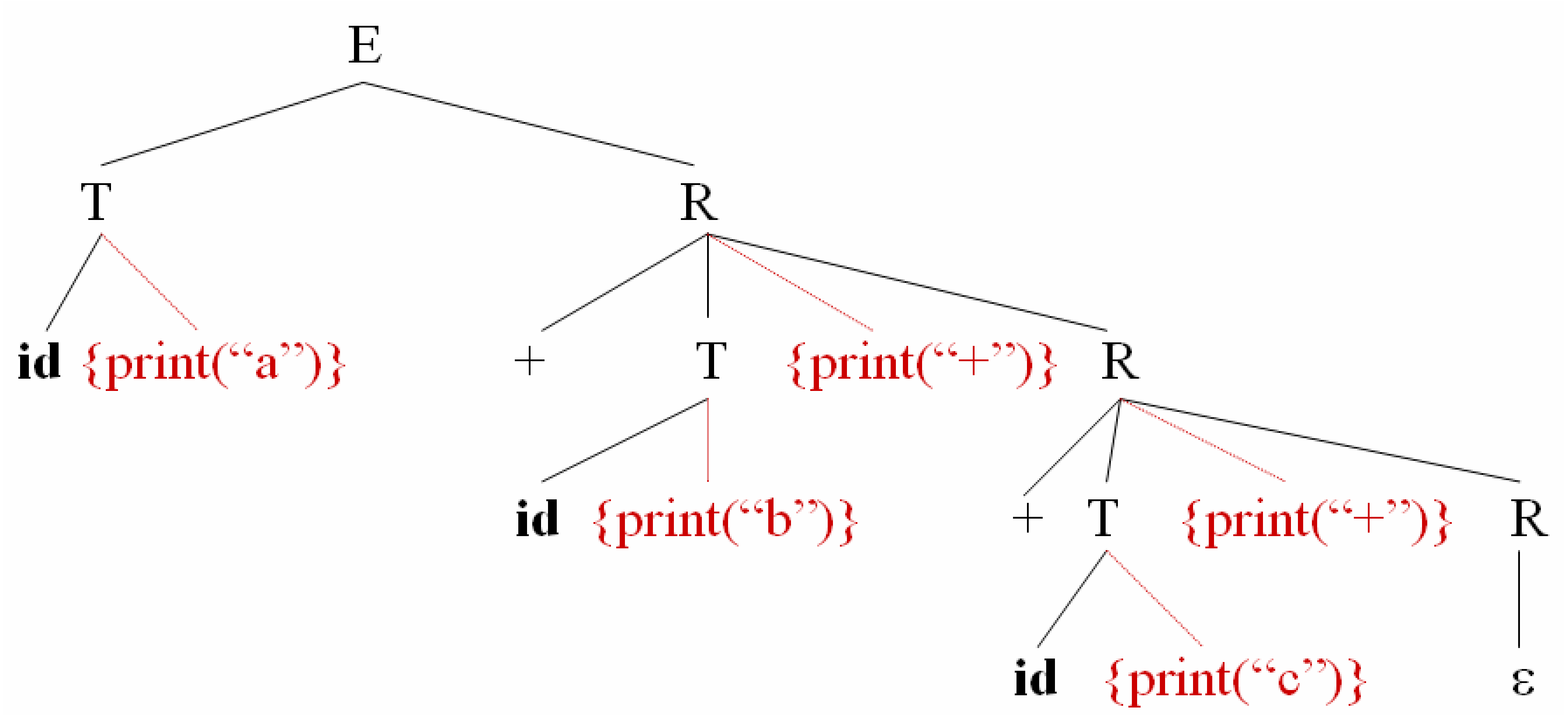
$R \rightarrow \varepsilon$

$T \rightarrow$ **id** { print(**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.

Inherited Attributes in Translation Schemes

- If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules:
  1. An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.
  2. A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
  3. 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).

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.

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);
}
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"); }
}
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"); }
}
```

a synthesized attribute (an output parameter)

an inherited attribute (an input parameter)