SYNTAX-DIRECTED TRANSLATION

Based on Chapter 5 of Aho, Lam, Sethi, Ullman:

Compilers: Principles, Techniques, & Tools

2nd Ed, Addison Wesley, 2007

Table of Contents

Syntax-Directed Translation Schemes

- **A syntax-directed translation scheme(SDT)** is a context-free grammar in which
  - attributes are associated with the grammar symbols
  - *semantic actions* are inserted within the right sides of productions
- Example (infix to postfix)

E-> T R

R -> addop T { print (addop.lexeme) } R1 | ε

T -> num { print(num.lexval) }

9 – 5 + 2
=> 9 5 – 2 +

Translation Schemes

- Implementation
  - Conceptually, by first building a parse tree and the performing the action in a left-to-right depth-first order
  - Practically, SDT's are implemented during parsing, without building a parse tree
  - Not all SDT's can be implemented during parsing

- Two Important Classes for Translation Schemes
  1. S-attributed definitions
  2. L-attributed definitions

  - attributes are available when an action refers to it
  - semantic rules in an SDD an be converted into n SDT with actions that are executed at the right time

Postfix Translation Schemes

- Postfix Translation Schemes
  - translation schemes for S-attributed definitions
  - create an action for each semantic rule
  - place the action at the end of the right side of the associated production

- Example 5.18 : Postfix SDT implementing the desk calculator

L -> E n	{ print(E.val); }
E-> E_1 + T	{ E.val = E_1.val + T.val; }
E -> T	{ E.val = T.val; }
T->T1 * F	{ T.val = T.val * F.val; }
T -> F	{ T.val = F.val; }
F-> (E)	{ F.val = E.val; }
F-> digit	{ F.val = digit.lexval; }

Parser-Stack Implementation of Postfix SDT

- synthesized attributes can be evaluated by **bottom-up** parser as input is being parsed
- keeps the values for the attributes on the parser stack(extended)
- when reduction occurs, the attribute values of the left side symbol of the production can be evaluated using the attribute values appearing on the stack for the right side symbols of the production

	state	***value***
	...	.....
	X	X.x
	Y	Y.y
top →	Z	Z.z
	....	.....

Semantic rule for A->XYZ

A.a = f(X.x, Y.y, Z.z)

implemented by

stack[ntop].val
=f(stack[top-2].val,stack[top-1].val,stack[top].val)

Parser-Stack Implementation of Postfix SDT

- Example : Desk calculator

Production	**Actions**
L -> E n	{ print(stack[top-1].val), top=top-1; }
E-> E_1 + T	{ stack[ntop].val=stack[top-2].val+stack[top].val; top=top-2; }
E -> T	
T->T1 * F	{stack[ntop].val=stack[top-2].val*stack[top].val; top=top-2; }
T -> F	
F-> (E)	{ stack[ntop].val = stack[top-1].val; top=top-2; }
F-> **digit**	

- ntop = top – r + 1 (r = | right side of the production |)
- when parser shifts a **digit** onto the stack, its attribute value(**digit**.lexval) is placed in stack[top].val

Parser-Stack Implementation of Postfix SDT

- Example: Desk calculator

3 * 5 + 4

state	val	input	production use
-	-	3 * 5 + 4 n	
d	3	* 5 + 4 n	
F	3	* 5 + 4 n	F -> digit
T	3	* 5 + 4 n	T -> F
T *	3 -	5 + 4 n	
T * d	3 - 5	+ 4 n	
T * F	3 - 5	+ 4 n	F -> digit
T	15	+ 4 n	T -> T * F
E	15	+ 4 n	E -> T
E +	15 -	4 n	
E + d	15 - 4	n	
E + F	15 - 4	n	F->digit
E + T	15 - 4	n	T->F
E	19	n	E->E+T
E n	19 -		
L	19		L->E n

SDT's with Actions Inside Productions

- Actions Inside Productions

  B -> X {*a*} Y

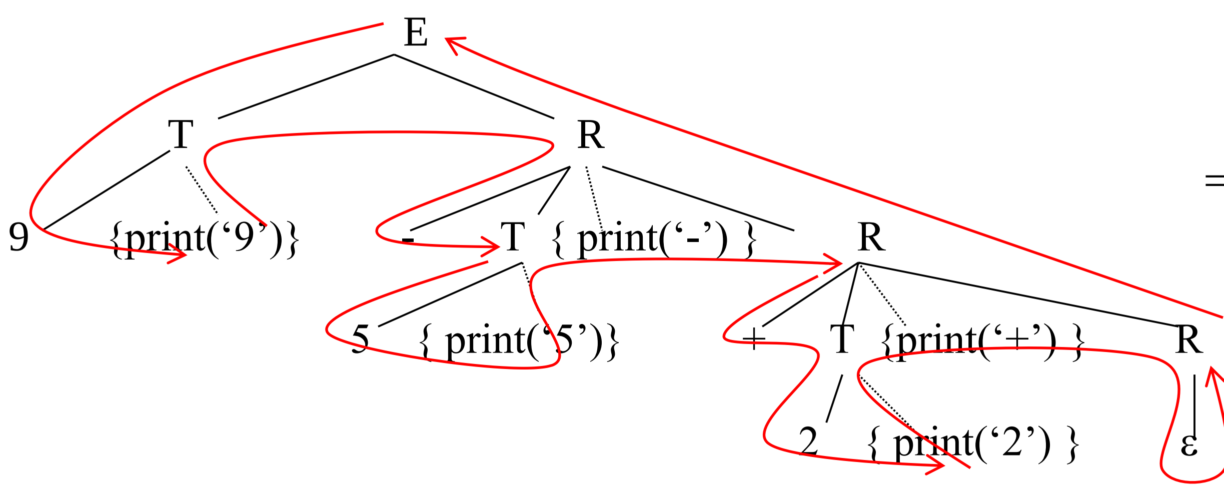
  - If the parse is bottom-up, then we perform action *a* as soon as X appears on the top of the parsing stack

- Example (revisit infix-to-postfix)

  E->T R

  R -> addop T { print (addop.lexeme) } R1 | ε

  T -> num { print(num.lexval) }



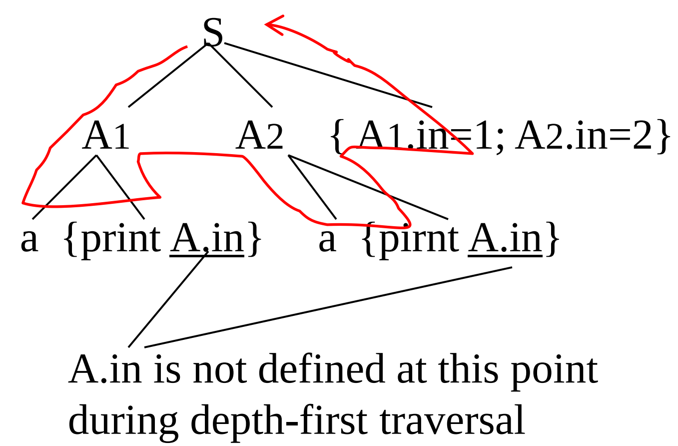
9 – 5 + 2
=> 9 5 – 2 +

SDT's for L-Attributed Definitions

- The rules for turning an L-attributed SDD into an SDT:
  1. Embed the action that computes the inherited attribute for a nonterminal *A* immediately before that occurrence of *A* in the body of the production
  2. Place the action for a synthesized attribute for the head of the production at the end of the body of that production
- Example

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

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

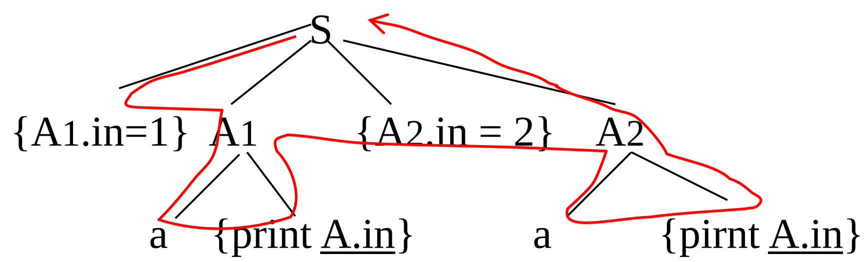


SDT's for L-Attributed Definitions

- Modified translation scheme

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

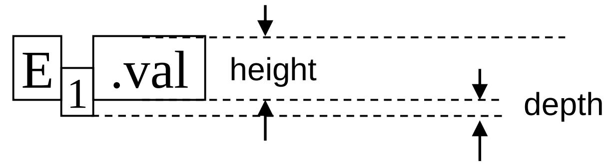
  $A \rightarrow a\ \{print(A.in)\}$



SDT's for L-Attributed Definitions

- Example 5.18: Typesetting mathematical formulas (e.g. TEX)

E sub 1 .val



- Syntax-directed definition

Production	Semantic Rules	
S -> B	B.ps = 10	
B -> B_1 B_2	B_1.ps = B.ps	
	B_2.ps = B.ps	
	B.ht = max(B_1.ht, B_2.ht)	
	B.dp = max(B_1.dp, B_2.dp)	
B -> B_1 **sub** B_2	B_1.ps = B.ps	
	B_2.ps = shrink(B.ps)	// e.g. B_2.ps = B.ps * 0.7
	B.ht = getHt(B_1.ht, B_2.ht, B.ps)	// e.g. B.ht = max(B_1.ht,B_2.ht-0.25*B.ps)
	B.dp = getDp(B_1.dp, B_2.dp, B.ps)	// e.g. B.dp = max(B_1.dp,B_2.dp+0.25*B.ps)
B -> **text**	B.ht = getHt(B.ps, **text**.lexval)	
	B.dp = getDp(B.ps, **text**.lexval)	

Inherited Attributes : ps
Synthesized Attributes: ht, dp

SDT's for L-Attributed Definitions

- Example 5.18 : Typesetting(cont.)
  - Translation Scheme

S -> { $B.ps = 10$; }
B

B -> {$B_1.ps = B.ps$; }
B_1 {$B_2.ps = B.ps$; }
B_2 {$B.ht = max(B_1.ht, B_2.ht)$;
$B.dp = max(B_1.dp, B_2.dp)$; }

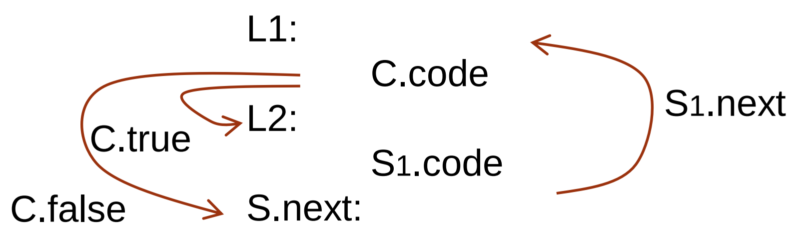
B -> {$B_1.ps = B.ps$; }
B_1
sub {$B_2.ps = shrink(B.ps)$; }
B_2 {$B.ht = getHt(B_1.ht, B_2.ht, B_2.ps)$;
$B.dp = getDp(B_1.dp, B_2.dp, B_2.ps)$;}

B -> text { B.ht = getHt(B.ps, text.lexval);
B.dp = getDp(B.ps, **text**.lexval); }

SDT's for L-Attributed Definitions

- Example 5.19: while statement
  - Grammar

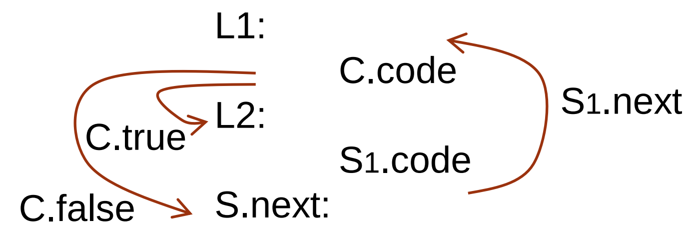
    S -> while (C) S_1
  - Intermediate Code Structure



  - Attributes
    ① S.next : inherited attribute – the label of the beginning of the code after S
    ② S.code : synthesized attribute – the intermediate code of the while statement
    ③ C.true : inheritance attribute – the label of the beginning of the code that must execute if C is true
    ④ C.false : inheritance attribute – the label of the beginning of the code that must execute if C is false

SDT's for L-Attributed Definitions

- Example 5.19: while statement (cont.)
  - Syntax Directed Definition



$S \to$ while (C) S_1

L1 = new label()
L2 = new label()
S_1.next = L1
C.false = S.next
C.true = L2
S.code = label L1 || C.code || label L2 || S_1.code

  - Translation Scheme

$S \to$ while	{ L1 = new label(); L2 = new label(); C.false = S.next; C.true = L2; }
(C)	{ S_1.next = L1; }
S_1	{ S.code = label L1 || C.code || label L2 || S_1.code; }

Bottom-up Parsing of L-Attributed SDD's

- Strategies

  Assume that A -> $\alpha\ \beta$, rule for action a

1. Embed actions (according to the rules at Slide 10)

   A -> α { a } β

2. Use marker nonterminal M to remove embedded actions and to add a rule M -> ε with associated rules, and to put the action at the end of rule M

   A -> α M β

   M -> ε {a'}

3. Evaluate inherited attributes
   - use inherited and synthesized attributes of maker nonterminal M
4. Implement copy rules using values in the parser stack

Bottom-up Parsing of L-Attributed SDD’s

- Removing Embedded Actions (without inherited attributes)
  - using maker nonterminals
  - Example : simple actions

E -> T R

R -> + T { print(‘+’)} R | - T {print(‘-’)} R | ε

T -> num {print(num.lexval) }

=>

E -> T R

R -> + T M R | - T N R | ε

T -> num {print(num.lexval) }

M-> ε { print(‘+’)}

N-> ε {print(‘-’)}

two translation schemes generate the same lang. and actions are performed in the same order

Bottom-up Parsing of L-Attributed SDD's

- Evaluation of Inherited Attributes : Copy Rules

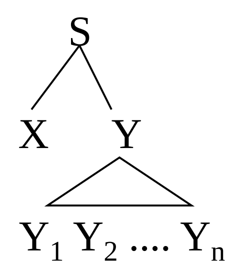
A -> XY   Y.in = X.s

X.s : a synthesized attribute of X

X.s는Y의 subtree가 parsing될 때 항상 stack에 존재한다.
즉,Y에 의해 inherit될 때 그 값을 stack에서 찾을 수 있다.

Y.in이 필요할 때 항상 X.s를 stack에서 찾아 사용할 수 있다.

....X Y_1 Y_2 ...Y_k →



Bottom-up Parsing of L-Attributed SDD's

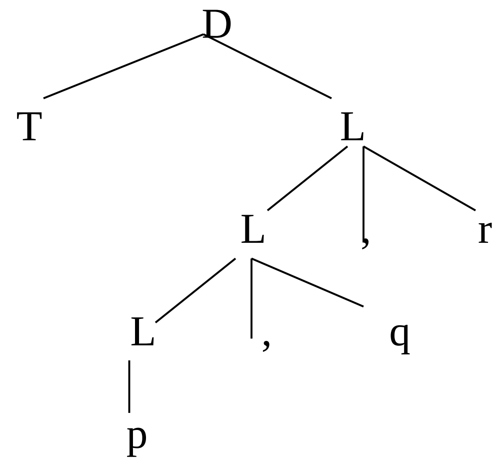
- Example (Type declaration)
  - Translation scheme

D-> T	{L.in = T.type; }	// copy rule
L		
T -> int	{T.type = integer; }	
T-> float	{T.type = float; }	
L ->	{L_1.in = L.in; }	// copy rule
L_1 , id	{ addtype(id.entry, L.in); }	
L -> id	{ addtype(id.entry, L.in); }	

Bottom-up Parsing of L-Attributed SDD's

- Example (Type declaration)
  - Parser Stack

Input	State	Production
float p,q,r	-	
p,q,r	float	
p,q,r	T	T->float
,q,r	T p	
,q,r	T L	L->id
q,r	T L ,	
,r	T L , q	
,r	T L	L->L , id
r	T L ,	
	T L , r	
	T L	L->L , id
	D	D->T L



stack[top-1]

stack[top-3]

Bottom-up Evaluation of Inherited Attributes

- Example (Type declaration)
  - Implementation for Type Declaration

D-> T L	
T -> int	{val[top].type = integer }
T-> float	{val[top].type = float }
L -> L_1 , id	{ addtype(id.entry, val[top-3].type) }
L -> id	{ addtype(id.entry, val[top-1].type) }

Bottom-up Parsing of L-Attributed SDD's

- Evaluation of Inherited Attributes : Non-copy Rules
  - Modify the action {a} to make {a'} as follows:

    $A \to \alpha \; \{\, a \,\} \; \beta$

    $\Rightarrow$

    $A \to \alpha \; \{\text{copy rules}\} \; M \; \{\text{copy rules}\} \; \beta$

    $M \to \varepsilon \; \{a'\}$

  - Copies, as inherited attributes of M, any attributes of A or symbols of α that action a needs
  - Computes attributes in {a'} in the same way as {a}, but makes those attributes be synthesized attributes of M
  - Copies, as inherited attributes of β, the synthesized attributes of M

Bottom-up Parsing of L-Attributed SDD's

- Evaluation of Inherited Attributes : Non-copy Rules
  - Example

    S -> aAC   C.i = f(A.s)

    - Embed actions

      S -> a A { C.i = f(A.s); } C
    - Marker nonterminal for embedded actions

      S -> aA { M.i = A.s; } M { C.i = M.s; } C

      M -> ε { M.s = f(M.i); }
    - Implement copy rules

      S -> aAMC

      M -> ε   { val[ntop] = f(val[top]); }

Bottom-up Parsing of L-Attributed SDD's

- Evaluation of Inherited Attributes
  - When the position of attribute cannot be predicted

Production	Semantic Rules
S -> aAC	C.i = A.s
S -> bABC	C.i = A.s
C -> c	C.s = g(C.i)

- C inherits the synthesized attribute A.s by a copy rule
- But, there may or may not be between A and C
- So, C.i is either val[top-1] or val[top-2]

Bottom-up Parsing of L-Attributed SDD's

- SDD using Marker Nonterminals

S -> aAC	C.i = A.s
S -> bABMC	M.i = A.s; C.i = M.s
C -> c	C.s = g(C.i)
M -> ε	M.s = M.i

- Syntax Directed Translation Scheme

S -> aA { C.i = A.s; } C

S -> bAB { M.i = A.s; } M {C.i = M.s; } C

C ->c { C.s = g(C.i); }

M -> ε { M.s = M.i; }

- Implementation using Bottom-up Parsing

S -> aAC	
S -> bABMC	
C -> c	val[ntop] = g(val[top-1])
M -> ε	val[ntop] = val[top-1]

Bottom-up Parsing of L-Attributed SDD's

- Example

```
S ->      { B.ps = 10; }
     B
B ->      {B1.ps = B.ps; }
     B1   {B2.ps = B.ps; }
     B2   {B.ht = max(B1.ht, B2.ht);
           B.dp = max(B1.dp, B2.dp); }

B ->      {B1.ps = B.ps; }
     B1
     sub  {B2.ps = shrink(B.ps); }
     B2   {B.ht = getHt(B1.ht, B2.ht, B2.ps);
           B.dp = getDp(B1.dp, B2.dp, B2.ps);}

B -> text { B.ht = getHt(B.ps, text.lexval);
            B.dp = getDp(B.ps, text.lexval); }
```

```
S -> L B
L -> ε       { L.ps = 10; }

B -> B1      { M.i = B.ps; }
     M       { B2.ps = M.s; }
     B2      {B.ht = max(B1.ht, B2.ht);
              B.dp = max(B1.dp, B2.dp); }
M -> ε       { M.s = M.i; }

B -> B1      { N.i = B.ps; }
     sub N   {B2.ps = N.s; }
     B2      { B.ht = getHt(B1.ht, B2.ht, B2.ps);
               B.dp = getDp(B1.dp, B2.dp, B2.ps);}
N -> ε       { N.s = shrink(N.i); }

B -> text    { B.ht = getHt(B.ps, text.lexval);
               B.dp = getDp(B.ps, text.lexval); }
```

Bottom-up Parsing of L-Attributed SDD's

- Implementation

Production	**Code Fragment**
S -> L B	
L -> ε	val[ntop].ps = 10
B -> B M B	val[ntop].ht = max(val[top-2].ht, val[top].ht)
	val[top].dp = max(val[top-2].dp, val[top].dp);
M -> ε	val[ntop].s = val[top-1].ps
B -> B sub N B	val[ntop].ht = getHt(val[top-3].ht, val[top].ht, val[top-1].ps)
	val[ntop].dp = getDp(val[top-3].dp, val[top].dp, val[top-1].ps)
N -> ε	val[ntop] = shrink(val[top-2])
B -> text	val[ntop] = getHt(val[top-1], val[top].lexval)