Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Sree Pranavi G

Compiler Design

Unit 3: SDD to SDT

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- L-attributed SDD to SDT
- SDT Implementation
- S-attributed SDD to SDT (Postfix SDT)

Translation by traversing a parse tree

- **Build parse tree and annotate.**
- **Build the parse tree, add actions and execute the actions in pre-order.**

Translation by traversing a parse tree

- **Build parse tree and annotate.**
- **Build the parse tree, add actions and execute the actions in pre-order.**

Translation during parsing

- Use a RDP with a function for each non-terminal.
- Generate code on the fly, using a RDP.
- Implement an SDT in conjunction with an LL parser.
- Implement an SDT in conjunction with an LR parser.

Conversion of L-attributed SDD to SDT

- Place the computation of inherited attributes for a non-terminal before that non-terminal appears in the right hand side of the production.

- Place the computation of synthesized attributes at the end of production.

Evaluation of L-attributed SDD

- L-attributed SDDs can have both synthesized attributes and inherited attributes.

Rule to be followed for evaluation

- Place the semantic rule corresponding to the inherited attributes of the non-terminal before the non-terminal appears on the right hand side of the production.

- Place the semantic rule corresponding to the synthesized attributes of the non-terminal at the end of the production.

Conversion of L-attributed SDD

Example

Production	Semantic Rule
T -> FT’	T’.ival = F.val; T.val = T’.val;
T ->*FT1’	T1’.ival = T.ival * F.val; T’.val = T1’.val;
T’ -> λ	T’.val = T’.ival;
F -> num	F.val = num.lexval

Conversion of L-attributed SDD

Example (contd.)

SDT

T -> F { T'.ival = F.val;} T' {T.val = T'.val;}

T' -> *F {T1'.ival = T'.ival * F.val;} T1' {T'.val =T1'.val;}

T' -> λ { T'.val = T'.ival;}

F -> num{ F.val = num.lexval; }

L-attributed SDD to SDT - Type Declarations

Production	Semantic Rule	Translation Scheme
$D \rightarrow T\ L$	*{ L.in = T.type;}*	$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ int	*{ T.type = integer; }*	$T \rightarrow$ int { T.type := 'integer' }
$T \rightarrow$ real	*{ T.type = float; }*	$T \rightarrow$ real { T.type := 'real' }
$L \rightarrow L_1$, id	*{ L_1.in= L.in; addType(id.entry, L.in); }*	$L \rightarrow$ { L1.in := L.in } L1, id { addtype(id.entry, L.in) }
$L \rightarrow$ id	*{ addType(id.entry, L.in); }*	$L \rightarrow$ id { addtype(id.entry, L.in) }

PES UNIVERSITY

L-attributed SDD to SDT - Intermediate Code Generation

Convert the L-attributed SDD to generate intermediate code for If statement to SDT -

Production	Semantic Rule
S → if(C) S1	*C.true = new label();* *C.false = S.next;* *S1.next =* *S.next;* *S.code = C.code || label(C.true) || S1.code*

L-attributed SDD to SDT

Translation Scheme -

S → if ({ C.true = new label (); C.false = S.next S1.next = S.next } C) S1 { S.code = C.code || label(C.true) || S1.code }

L-attributed SDD to SDT

Convert the L-attributed SDD to generate intermediate code for while statement to SDT :

Production	Semantic Rule
S → while (C) S1	*begin = new label();* *C.true = new label();* *C.false = S.next* *S1.next = begin* *S.code = label(begin) || C.code || label(B.true) ||* *S1.code || gen('goto' label(begin));*

Translation Scheme -

S → while ({ begin = new label (); C.true = new label (); C.false = S.next; S1.next = begin; } C) S1 { S.code = label(begin) || C.code || label(C.true) || S1.code || gen('goto' label(begin)); }

SDT

Implementation

There are two ways to implement a SDT scheme :

1. After Parsing - Work with the output of Parser (Parse Tree)
2. During Parsing

SDT Implementation - After Parsing

- Ignoring the actions, parse the input and produce a parse tree as a result.
- Examine each interior node N.
- Add additional children to N for the actions.
- Perform a preorder traversal of the tree, and as soon as a node labelled by an action is visited, perform that action.

SDT Implementation - After Parsing

- **Consider the following translation scheme.**

  S ⟶ ER

  R ⟶ *E { print("*"); } R | ε

  E ⟶ F + E { print("+"); } | F

  F ⟶ (S) | id { print(id.value); }

  Here id is a token that represents an integer and id.value represents the corresponding integer value.

  For an input 2 * 3 + 4, this translation scheme prints ________________.

SDT - Infix to Postfix

Production	Semantic Rule
E ⟶ E1 + T	
E ⟶ T	
T ⟶ T1 * F	
T ⟶ F	
F ⟶ num	
F ⟶ id	

SDT - Infix to Postfix

Production	Semantic Rule
E ⟶ E1 + T	*E ⟶ {printf("+");} E1+ T*
E ⟶ T	*E ⟶ T*
T ⟶ T1 * F	*T ⟶ {printf("*");} T1* F*
T ⟶ F	*T ⟶ F*
F ⟶ num	*F ⟶ num {printf("%d", num.lexval);}*
F ⟶ id	*F ⟶ id {printf("%d", id.lexval);}*

Evaluation of S-Attributed Definitions

- Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.
- The parser keeps the values of the synthesized attributes in its stack.
- Whenever a reduction $A \longrightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.
- Thus, a translator for an S-Attributed Definition can be simply implemented by extending the stack of an LR-Parser.

Extending a Parser Stack

- Extra fields are added to the stack to hold the values of synthesized attributes.
- In the simple case of just one attribute per grammar symbol the stack has two fields - state and val.
- The current top of the stack is indicated by the pointer top.

Parser Stack

- **Synthesized attributes are computed just before each reduction :**
  - **Before the reduction A → XYZ is made, the attribute for A is computed :**

    A.a := f (s[top].val, s[top −1].val,

    s[top − 2].val).

state	val
Z	*Z.x*
Y	*Y.x*
X	*X.x*
…	…

Postfix SDT scheme for simple desk calculator

E $\longrightarrow$ E1 +T	{stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }
E $\longrightarrow$ T	
T $\longrightarrow$ T1 * F	{stack[top-2].val = stack[top-2].val * stack[top].val; top = top - 2;}
T $\longrightarrow$ F	
F $\longrightarrow$ (E)	{stack[top-2].val = stack[top-1].val; top = top - 2;}
F $\longrightarrow$ digit	

Implementing SDT Scheme during LR Parsing

- Introduce a Marker Non-terminal in place of each embedded action.

- There is one production for each Marker M,

  $M \longrightarrow \lambda$

Example :

$A \longrightarrow$ alpha {a} beta {b}

would convert to

$A \longrightarrow$ alpha M beta {b}

$M \longrightarrow \lambda$ {a}

Implementing SDT Scheme during LR Parsing

- For Example consider the following SDT scheme to generate Intermediate code :

P → {S.next = new label(); } S { P.code = S.code || label(S.next); }

S → id = num ; { S.code = gen(id.name "=" num.lexval); }

S → while ({ begin = new label (); C.true = new label (); C.false = S.next; } C) { S1.next = begin; } S1 { S.code = label(begin) || C.code || label(C.true) || S1.code || gen("goto" label(begin)); }

C → id_1 rel id_2 ; { C.addr = new Temp();
C.code = gen (C.addr "=" id1.name rel.op id2.name)
|| gen ("if" C.addr "goto" label(C.true)) || gen("goto" label(C.false)); }

Implementing SDT Scheme during LR Parsing

Introducing a Marker Non Terminal for every embedded action…

P ⟶ X S { P.code = S.code || label(S.next); }
X ⟶ λ {S.next = new label(); }

S ⟶ id = num ; { S.code = gen(id.name "=" num.lexval); }

S ⟶ while (M C) N S { S.code = label(begin) || C.code || label(C.true) || S1.code ||
gen("goto" label(begin)); }
M ⟶ λ {begin = new label (); C.true = new label (); C.false = S.next; }
N ⟶ λ {S1.next = begin;}

C ⟶ id_1 rel id_2 ; { C.addr = new Temp();
C.code = gen (C.addr "=" id1.name rel.op id2.name)
|| gen ("if" C.addr "goto" label(C.true))
|| gen("goto" label(C.false)); }

Implementing SDT scheme during LR Parsing

Parser Stack Structure

Stack record	*A*	*Synthesized attributes of A*
Record of Marker A	Inherited attributes of A	

Note : We perform general style of bottom-up parsing - shift-reduce parsing. A - non terminal

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$	while (a>b) a = 0; $	reduce using $X \longrightarrow \lambda$
$ X X, S.next = new label(); $	while (a>b) a = 0; $	Shift while
$ X while while X, S.next = new label(); $	(a>b) a = 0; $	Shift (

X → λ { s[top].next = new label();}

Implementing SDT scheme during LR Parsing

- **Example continued…**

<table>
<tr><th>Stack</th><th>Input Buffer</th><th>Action</th></tr>
<tr><td>$ X while (
<table><tr><td>(</td></tr><tr><td>while</td></tr><tr><td>X, S.next = new label();</td></tr><tr><td>$</td></tr></table></td><td>a>b) a = 0; $</td><td>reduce using M → λ</td></tr>
<tr><td>$ X while (M
<table><tr><td>top</td><td>M, begin = new label();, C.true= new label();
C.false = S.next</td></tr><tr><td>-1</td><td>(</td></tr><tr><td>-2</td><td>while</td></tr><tr><td>-3</td><td>X, S.next = new label();</td></tr><tr><td></td><td>$</td></tr></table></td><td>a>b) a = 0; $</td><td>Shift id
(id.name = a)</td></tr>
</table>

M → λ { s[top].begin = new label();
s[top].true = new label();
s[top].false = s[top-3].next;}

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M id	>b) a = 0; $	Shift rel (rel.op = >)

id, id.name = a
M, begin = new label(), C.true= new label(); C.false = S.next
(
while
X, S.next = new label();
$

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M id rel	b) a = 0; $	Shift id(id.name = b)

rel , rel.op = “>”
id, id.name = a
M, begin = new label(); C.true= new label(); C.false = S.next
(
while
X, S.next = new label();
$

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M id rel id	) a = 0; $	reduce using C ⟶ id rel id

top	id, id.name = b
-1	rel , rel.op = “>”
-2	id, id.name = a
	M, begin = new label(); C.true= new label(); C.false = S.next
	(
	while
	X, S.next = new label();
	$

```
C ⟶ id rel id { s[top-2].addr = new Temp();
s[top-2].code = gen(s[top].addr “=” s[top-2].name s[top-1].op s[top].name)
|| gen(“if” s[top].addr “goto” label(s[top-3].true)
|| gen(“goto” label(s[top-3].false);

top = top -2; }
```

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M C	) a = 0; $	Shift)

C, C.addr = new Temp(); C.code
M, begin = new label(); C.true= new label(); C.false = S.next
(
while
X, S.next = new label();
$

Implementing SDT scheme during LR Parsing

- **Example continued…**

<table>
<tr><th>Stack</th><th>Input Buffer</th><th>Action</th></tr>
<tr><td>$ X while (M C)

<table>
<tr><td></td><td>)</td></tr>
<tr><td></td><td>C, C.addr = new Temp(); C.code</td></tr>
<tr><td></td><td>M, begin = new label(); C.true= new label(); C.false = S.next</td></tr>
<tr><td></td><td>(</td></tr>
<tr><td></td><td>while</td></tr>
<tr><td></td><td>X, S.next = new label();</td></tr>
<tr><td></td><td>$</td></tr>
</table>

</td><td>a = 0; $</td><td>reduce using N → λ</td></tr>
</table>

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M C)	a = 0; $	Shift id (id.name = a)

top	N, S.next = begin
-1	)
-2	C, C.addr = new Temp(); C.code
-3	M, begin = new label(); C.true= new label(); C.false = S.next
	(
	while
	X, S.next = new label();
	$

N → λ {s[top].next = s[top-3].begin;}

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M C) N id	= 0; $	Shift =

id, id.name = a
N, S.next = begin
)
C, C.addr = new Temp(); C.code
M, begin = new label(); C.true= new label(); C.false = S.next
(
while
X, S.next = new label();
$

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M C) N id =	0; $	Shift num (num.lexval = 0)

=
id, id.name = a
N, S.next = begin
)
C, C.addr = new Temp(); C.code
M, begin = new label(); C.true= new label(); C.false = S.next
(
while
X, S.next = new label();
$

Implementing SDT scheme during LR Parsing

- Example continued…

Stack	Input Buffer	Action
$ X while (M C) N id = num	; $	Shift ;

num, num.lexval = 0
=
id, id.name = a
N, S.next = begin
)
C, C.addr = new Temp(); C.code
M, begin = new label(); C.true= new label(); C.false = S.next
(
while
X, S.next = new label();
$

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M C) N id = num ;	$	reduce using S → id = num ;

top	;
-1	num, num.lexval = 0
-2	=
-3	id, id.name = a
	N, S.next = begin
	)
	C, C.addr = new Temp(); C.code
	M, begin = new label(); C.true= new label(); C.false = S.next
	(
	while
	X, S.next = new label();
	$

S → id = num ; { s[top - 3].code =
gen(s[top-3].name "=" s[top-1].lexval); }

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X while (M C) N S	$	reduce using S ⟶ while(MC)NS

top	S, S.code
-1	N, S.next = begin
-2	)
-3	C, C.addr = new Temp(); C.code
-4	M, begin = new label(); C.true= new label(); C.false = S.next
-5	(
-6	while
	X, S.next = new label();
	$

```
S ⟶ while(MC)NS {
    s[top-6].code = label(s[top-4].begin ||
s[top-3].code || label(s[top-4].true || s[top].code ||
gen("goto" label(s[top-4].begin));

    top = top - 6;
    }
```

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ X S	$	reduce using $P \rightarrow X\ S$

top	S, S.code
-1	X, S.next = new label();
	$

```
P → S {
s[top-1].code = s[top].code || label(s[top-1].next);

top = top - 1;
}
```

Implementing SDT scheme during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ P P, P.code $	$	Parsing Successful

Implementing SDT Scheme during LR Parsing

Semantic Actions written in terms of stack operations.

P → X S {s[top-1].code = s[top].code || label(s[top-1].next); top = top - 1; }
X → λ {s[top].next = new label();}

S → id = num ; {s[top - 3].code = gen(s[top-3].name "=" s[top-1].lexval);}
S → while (M C) N S { s[top-6].code = label(s[top-4].begin || s[top-3].code ||
label(s[top-4].true || s[top].code
|| gen("goto" label(s[top-4].begin)); top = top - 6; }
M → λ {s[top].begin = new label(); s[top].true = new label();
s[top].false = s[top-3].next;}
N → λ {s[top].next = s[top-3].begin;}

C → id_1 rel id_2 ; { s[top-2].addr = new Temp();
s[top-2].code = gen(s[top].addr "=" s[top-2].name s[top-1].op s[top].name)
|| gen("if" s[top].addr "goto" label(s[top-3].true)
|| gen("goto" label(s[top-3].false); top = top -2; }

Example 2 :

$S \rightarrow$ if ({ C.true = new label (); C.false = S.next; } C) { S1.next = S.next; } S1 { S.code = C.code || label(C.true) || S1.code }

Run over the input string if(x>y) x= 5; and provide all semantic actions in terms of stack operations.

Implementing SDT scheme during LR Parsing

Example 3 :

T -> F { T'.ival = F.val;} T' {T.val = T'.val;}

T' -> *F {T1'.ival = T'.ival * F.val;} T1' {T'.val =T1'.val;}

T' -> λ { T'.val = T'.ival;}

F -> num{ F.val = num.lexval; }

Run over the input string 3*5 and provide all semantic actions in terms of stack operations.

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu