Implementation of Syntax-Directed Translators

Introduction

- The output defined by a syntax directed translation scheme is independent of any kind of parser used to construct the parse tree
- A syntax directed translation scheme provides a method for describing an input-output mapping
- And that description is independent of any implementation
  - It is easy to modify

Implementation

- After a syntax directed translation scheme is written, next task is to convert it into a program that implements the input-output mapping.
- Requirements
  - A bottom up parser for the grammar
  - The parser needs to be augmented with some mechanism for computing the translations

Implementation

- To compute the translation at a node A associated with a production

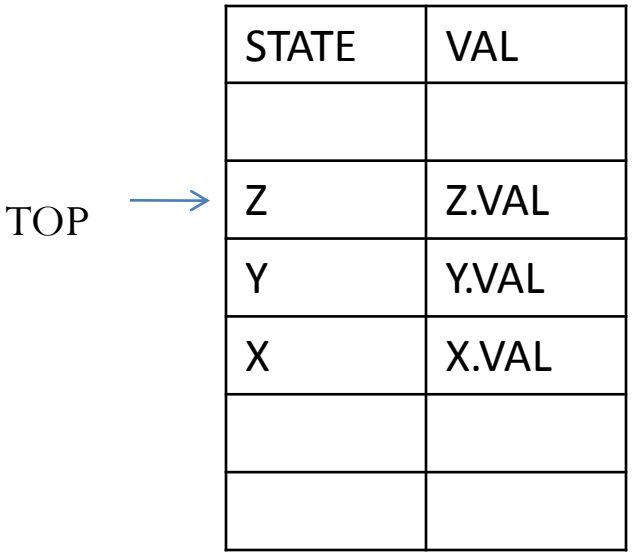
  $A \rightarrow XYZ$

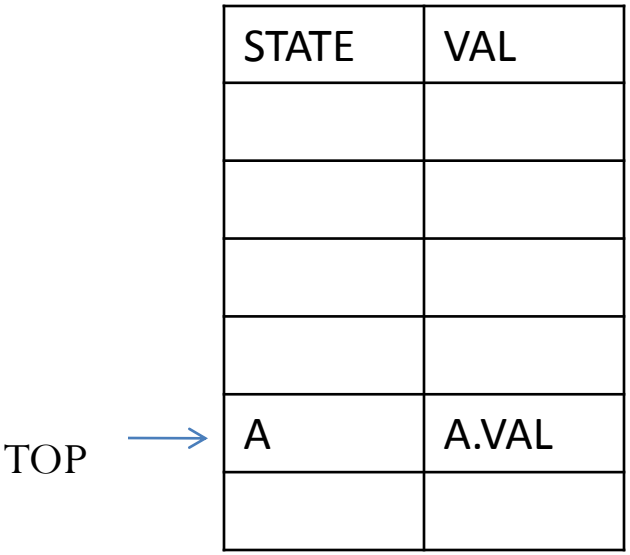
- We need the values of a translations associated with the nodes labeled X,Y, and Z
- These nodes will be root of sub-trees in the forest representing the partially constructed parse tree
- The nodes X, Y, and Z becomes the children of the node A after reduction by $A \rightarrow XYZ$
- Once the reduction has occurred we do not need the translations of X,Y, and Z any longer

Implementation

- Consider the parser stack as an array pair – STATE and VAL.
- Each STATE entry is a pointer to the LR(1) parsing table .
- If STATE [i] is the symbol E, the VAL[i] holds the translation for E.VAL associated with the parse tree node corresponding to E
- TOP is a pointer to the top of the stack.
- Let A$\rightarrow$XYZ be a production
  - Before XYZ is reduced to A

Implementation





Example

- use a syntax-directed translation scheme to specify a “desk calculator” program
- then this scheme is implemented by a bottom up parser
  - that invokes program fragments to compute the semantic actions
  - E.g., input expression 23*5+4$ the output of the program should be the value 119

Grammar for a desk calculator

$S \rightarrow E\$$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow I$

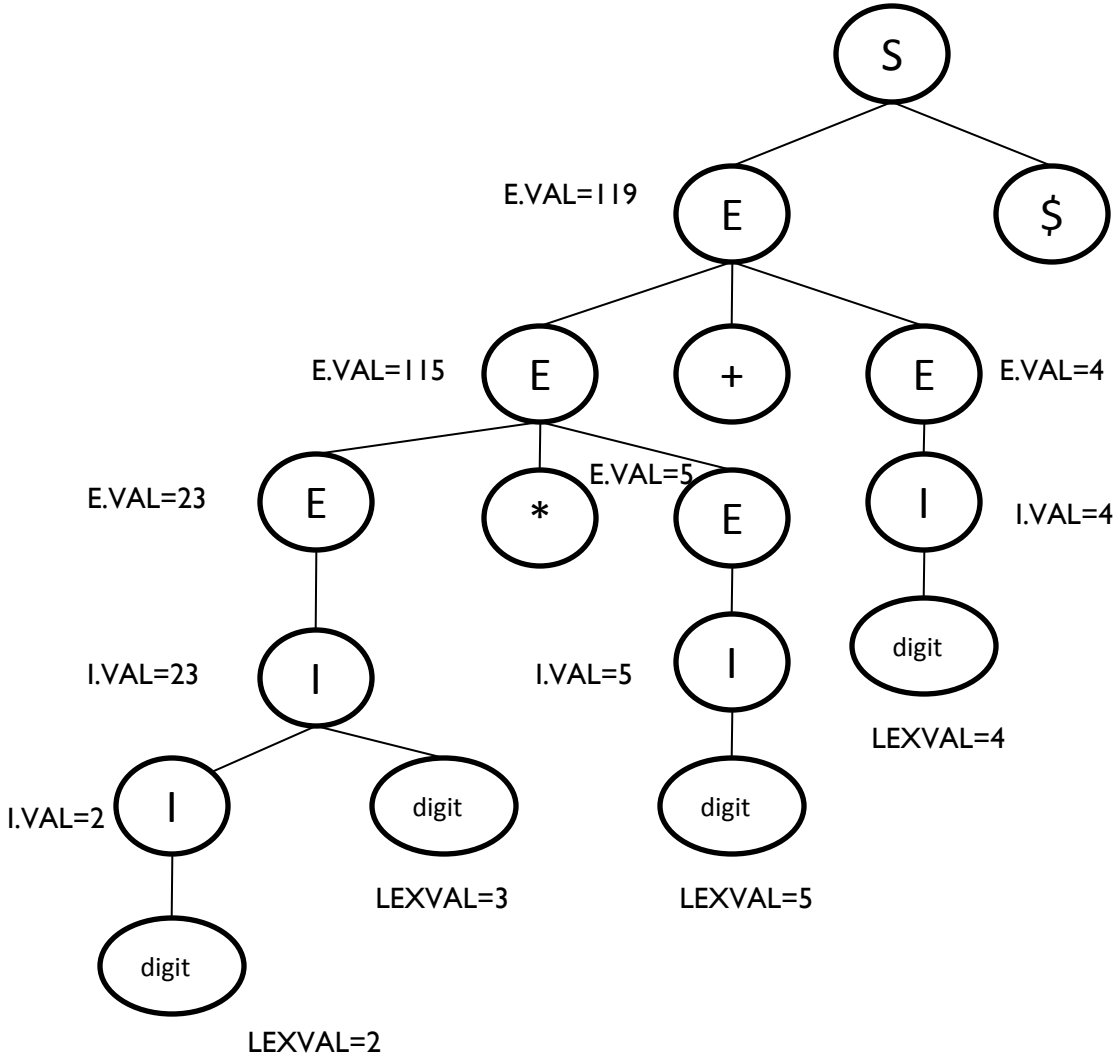
$I \rightarrow I\ digit$

$I \rightarrow digit$

- With non-terminals E and I, associate integer valued attribute VAL to denote the numerical value of the expression or integer represented by a node of the parse tree labeled E or I.
- With terminal **digit** we associate attribute LEXVAL which is returned by the lexical analyzer as (digit, LEXVAL).

Production	Semantic Rule
$S \rightarrow E\$$	{ Print E.VAL }
$E \rightarrow E1 + E2$	{ E.VAL := E1.VAL + E2.VAL}
$E \rightarrow E1 * E2$	{ E.VAL := E1.VAL * E2.VAL}
$E \rightarrow (E1)$	{ E.VAL := E1.VAL}
$E \rightarrow I$	{ E.VAL := I.VAL }
$I \rightarrow I$ digit	{I.VAL := 10*I.VAL + digit }
$I \rightarrow$ digit	{ I.VAL := LEXVAL }

Parse tree with translations



Implementation of a desk calculator

- To implement the semantic actions we cause the parser to execute the program fragments just before making the appropriate reduction

Production	Program Fragment
S → E$	Print VAL[TOP]
E →E1 + E2	VAL [TOP]:= VAL[TOP] + VAL[TOP-2]
E →E1 * E2	VAL [TOP]:= VAL[TOP] * VAL[TOP-2]
E → (E)	VAL[TOP] := VAL[TOP-1]
E →I	NONE
I →I digit	VAL[TOP] := 10*VAL[TOP]+LEXVAL
I →digit	VAL[TOP] := LEXVAL

Sequence of moves

	Input	STATE	VAL	Production used
(1)	23*5+4$	-	-	
(2)	3*5+4$	2	-	
(3)	3*5+4$	I	2	I→digit
(4)	*5+4$	I3	2_	
(5)	*5+4$	I	(23)	I→I digit
(6)	*5+4$	E	(23)	E→I
(7)	5 + 4$	E *	(23)_	
(8)	+4$	E*5	(23)_ _	
(9)	+ 4$	E * I	(23) _ 5	I→ digit
(10)	+ 4$	E * E	(23) _ 5	E→I
(11)	+ $$	E	(115)	E→E*E
(12)	4$	E +	(115) _	
(13)	$	E + 4	(115) _ _	
(14)	$	E + I	(115) _ 4	I→digit
(15)	$	E + E	(115) _ 4	E→I
(16)	$	E	(119)	E→E+I
(17)	_	E$	(119) _	
(18)	_	S	_	S→ E$

Sequence of moves

- Consider the sequence of events on seeing the input symbol 2
- In the first move the parser shifts the state corresponding to the token digit whose LEXVAL is 2 onto the stack
- Second move the parser reduces by the production I $\rightarrow$digit and invokes the semantic action I.VAL=LEXVAL
- The program fragment implementing this semantic action causes the VAL field of the stack entry for digit to acquire the value 2

Intermediate code

- In many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine language
  - Hence called an intermediate code or intermediate text

Intermediate codes

- Four kinds of intermediate code often used in compilers are

1. Postfix notation
2. Syntax trees
3. Quadruples
4. triples

Postfix notation

- In general if e_1 and e_2 are any postfix expressions, and θ is any binary operator the result is $e_1 e_2 \theta$
- No parenthesis are needed in postfix notation because the arity (number of arguments) of the operators permits only one way to decode a postfix expression

Examples

1. (a+b)*c in postfix notaion is ab+c*
2. a*(b+c) is abc+*
3. (a+b)*(c+d) is ab+cd+*

- If we know the arity of each operator then we can uniquely decipher any postfix expression by scanning from the either end

Example

- ab+c*
  - The right-hand * says that there are two arguments to its left
  - Since the next to the rightmost symbol is c, a simple operand,
    - we know c must be the second operand of *
  - Continuing to the left, we encounter the operator +.
    - We now know the sub-expression ending in + makes up the first operand of *
  - Continuing I this way we deduce ab+c* is parsed as (((a, b)+),c)*

Evaluation of Postfix expression

- A stack is used –hardware or in software
- General strategy is to scan the postfix code left to right
  - Push each operand onto the stack
  - If we encounter a k-ary operator,
    - its first argument will be k-1 positions below the top on the stack
    - Its last argument will be at the top
  - Then apply the operator to the top k values on the stack
  - These values are then popped and the result of applying the k-ary operator is pushed onto the stack

example

- ab+c* and values of a,b,c are 1,3 and 5 respectively
- The evaluation of 13+5* is perform using the following actions

1. Stack 1
2. Stack 3
3. Add the two topmost elements, pop them off the stack, and then stack the result, 4
4. Stack 5
5. Multiply the two topmost elements, pop them off the stack, and then stack the result, 20

The value on top of the stack at the end is the value of the entire expression

Syntax-directed translation to postfix code

- Syntax-Directed Translation for infix-postfix translation
- Production semantic action

$E \rightarrow E^{(1)}$ **op** $E^{(2)}$	E.CODE:=$E^{(1)}$.CODE ||$E^{(2)}$. CODE|| ' **op**'
$E \rightarrow (E^{(1)})$	E.CODE:=$E^{(1)}$.CODE
$E \rightarrow$ **id**	E.CODE:= **id**

Implementation of infix-postfix translation

Production	Program Fragment
$E \rightarrow E^{(1)}$ **op** $E^{(2)}$	{**print op**}
$E \rightarrow (E^{(1)})$	{}
$E \rightarrow$ **id**	{**print id**}