Compiler Construction

Syntax Directed Translation (SDT)

Mr. Usman Wajid

usman.wajid@nu.edu.pk

National University
of Computer & Emerging Sciences

What is a Syntax-directed Translation?

Syntax-directed Translation

The grammar together with semantic action is called syntax-directed translation (SDT)

- It is an extension of context free grammars (CFGs)
- Consider the following CFG,
  A $\rightarrow \alpha$ + { action } = SDT
- Semantic action or translation or semantic rule or action is the same
- We can attach semantic actions for grammar for performing different tasks, such as,
  1. To store / retrieve type information in symbol table
  2. To perform consistency checks like type checking, parameter checking, etc
  3. To issue error messages
  4. to build syntax trees
  5. to generate intermediate or target code

SDD vs SDT

There are two ways of defining semantic rules:

Syntax-directed Definition (SDD)

frees the user to explicitly specify the order of evaluation

- It is a high-level language specification or translation
- It hides many implementation details

Syntax-directed Translation (SDT)

It specifies the order by itself in which semantic rules are to be evaluated

- They allow some implementation details to be shown

The names SDD or SDT are sometimes used interchangeably but the basic difference between the two is in specifying the evaluation order

Attribute grammars

- SDTs are also called attribute grammars
- Each grammar symbol is assigned certain attributes
- Each production is augmented with semantic rules which are used to define attribute values

Attribute for Grammar Symbols

- It is a generalization of context-free grammar.
- Each grammar symbol has associated attributes.
- Parse tree nodes act like records.
- Attributes are like fields within these records.
- A grammar symbol can have ”n” attributes.
- Attribute types can include: type, value, address, pointer, or string.

Attribute Types

Synthesized Attribute

The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

$A \rightarrow XYZ \ \{A \bullet s = X \bullet s + Y \bullet s + Z \bullet s\}$

Then ."s" is called the synthesized attribute.

Inherited Attribute

The value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node.

$A \rightarrow XYZ \ \{ Y \bullet i = A \bullet i + X \bullet i + Z \bullet i\}$

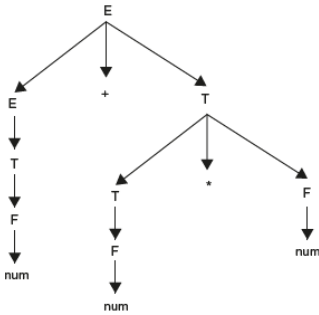
Then "•i" is called the inherited attribute.

Writing Syntax-Directed Translation Example1

- A semantic rule is combined with parser to carryout an additional task such as evaluating expression
- The bottomup parser gives us the order of reductions
- such as num to F, F to T ,..., E + T to E
- Whenever reduction occur, there should be a corresponding action to evaluate an expression
- For instance, we assume that whenever reduction occurs, the corresponding action is automatically carried out.

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow num$

input: 1 + 2 * 3

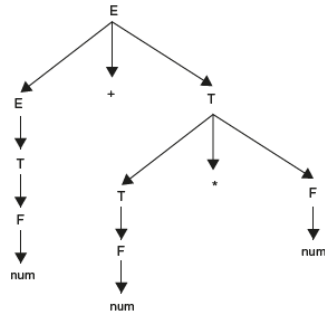


Writing Syntax-Directed Translation Example continued

- Bottom-up parser on reading symbol ”1” gets the token ”num” from lexical analyzer
- The parser reduces ”num” to ”F” by using rule F → num
- to evaluate an expression, a lexeme value of token ”num” is also required
- It can be stored as an attribute with grammar symbol
- An attribute is attached to grammar symbol with a dot ”.”

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow num$

input: 1 + 2 * 3

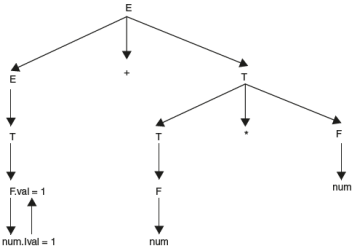


Writing Syntax-Directed Translation Example continued

- Assume ".val" attribute with each grammar symbol
- Hence, grammar symbols will become E.val, T.val, F.val
- When parser reduces "num" to F, an attribute value needs to be stored at node F
- Hence, we attach semantic action as {F.val = num.lval;}
- where num.lval is a lexeme value of token num.
- num.lval can be any value such as "1"

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow num$

input: 1 + 2 * 3



Writing Syntax-Directed Translation Example continued

- Next reduction is F to T
- Hence, the corresponding semantic action will be

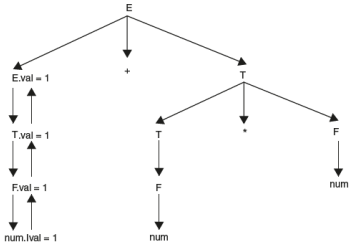
  {T.val = F.val}

- The same thing happens from T to E and corresponding action will be

  {E.val = T.val}

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow num$

input: 1 + 2 * 3

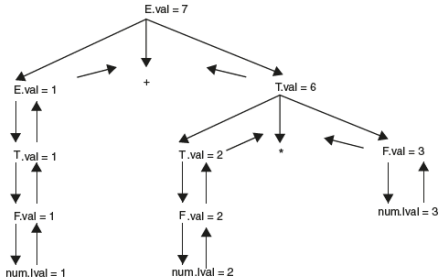


Writing Syntax-Directed Translation Example continued

- Hence, SDT defined:

$E \rightarrow E_1 + T$	$\{E\bullet val = E_1\bullet val + T\bullet val\}$
$E \rightarrow T$	$\{E\bullet val = T\bullet val\}$
$T \rightarrow T_1 * F$	$\{T\bullet val = T_1\bullet val * F\bullet val\}$
$T \rightarrow F$	$\{T\bullet val = F\bullet val\}$
$F \rightarrow id$	$\{F\bullet val = num\bullet lval\}$

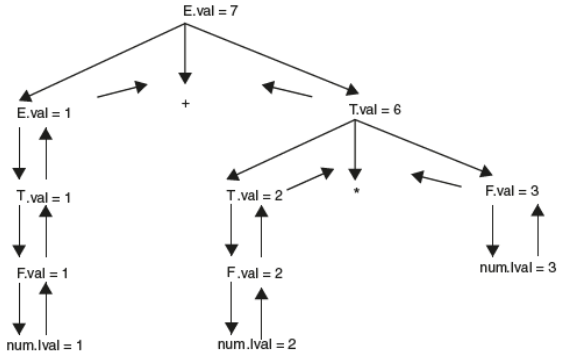
num + num * num
F + num * num
T + num * num
E + num * num
E + F * num
E + T * num
E + T * F
E + T
E



Writing Syntax-Directed Translation Example continued

- Hence, SDT defined:

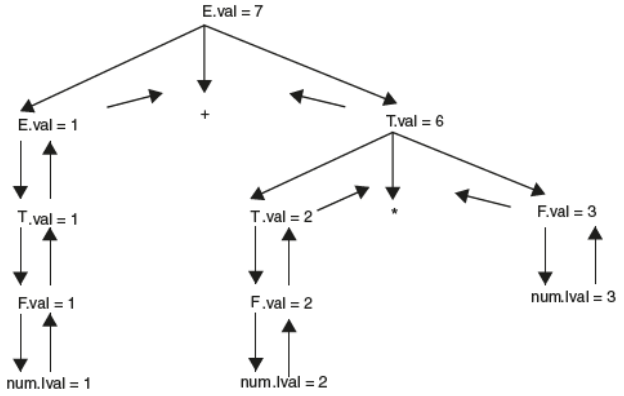
$E \rightarrow E_1 + T$	$\{E\bullet val = E_1\bullet val + T\bullet val\}$
$E \rightarrow T$	$\{E\bullet val = T\bullet val\}$
$T \rightarrow T_1 * F$	$\{T\bullet val = T_1\bullet val * F\bullet val\}$
$T \rightarrow F$	$\{T\bullet val = F\bullet val\}$
$F \rightarrow id$	$\{F\bullet val = num\bullet lval\}$



Writing Syntax-Directed Translation Example continued

Annotated or Decorated Parse Tree

The parse tree that shows attribute values at each node is called annotated or decorated parse tree



Writing SDT

SDT involes the following steps:

1. Define the grammar
   - Generate a parse tree for the sample input
2. Decide what to compute
   - such as evaluate an expression, generate intermediate or target code, type checking, etc
3. Augment semantic actions based on expected output

Writing SDT Example2

- Write an SDT for converting infix expressions to postfix form
- Given grammar:
  $E \rightarrow E + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow id$
- sample input: a + b * c
- expected output: a b c * +

Writing SDT Example2 continued

- sample input: a + b * c
- sample output: a b c * +

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow id$

- Hence, SDT defined:

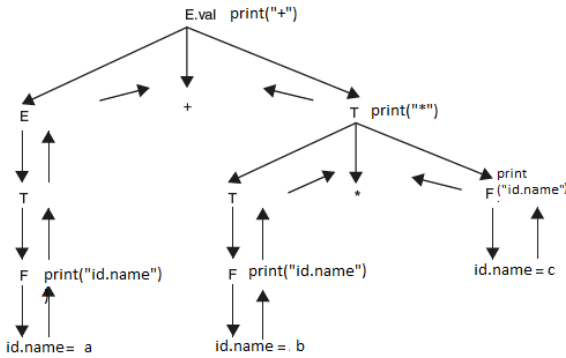
$E \rightarrow E + T$ {print("+");}

$E \rightarrow T$ {}

$T \rightarrow T * F$ {print("*");}

$T \rightarrow F$ {}

$F \rightarrow id$ {print(id.name);}



Writing SDT Example3 (home practice)

- Write an SDT for converting infix expression to prefix form
- Given grammar:

  $E \rightarrow E + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow id$

- sample input: a + b * c
- expected output: + a * b c

Writing SDT for simple type checker (Example4)

To Write SDT a simple type checker, assume:

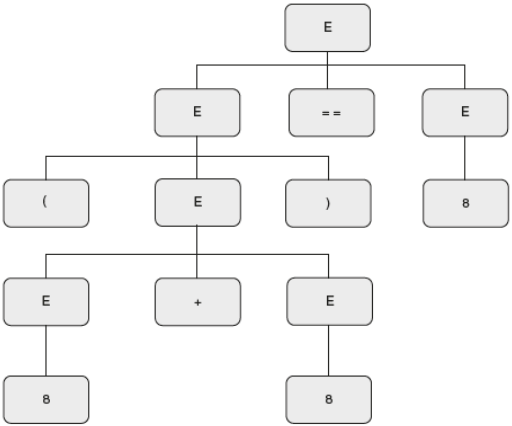
1. int — for integer literals (like 8)
2. bool — for Boolean literals (true, false)
3. err — for any type error

Writing SDT for simple type checker continued

- Grammar:

  E → E+ E
  E → E = = E
  E → true
  E → false
  E → num
  E → (E)

- Sample input: (8 + 8) = 8



Writing SDT for simple type checker continued

- Hence, SDT defined:

  $E_1 \rightarrow E_2 + E_3$ {if(E_2•type = =int and E_3•type = =int) then E_1•type = int else E_1•type=error;}

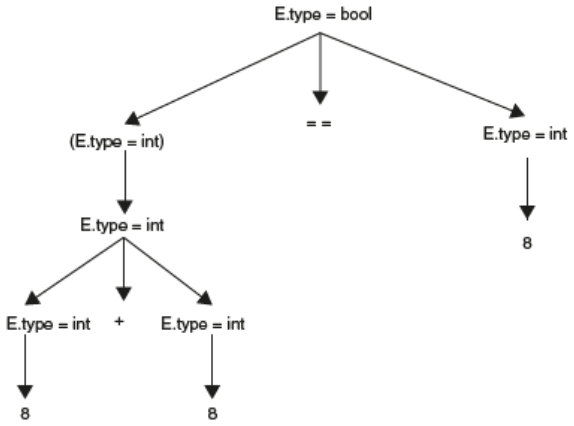
  $E_1 \rightarrow E_2 = = E_3$ {if(E_2•type = = E_3•type) then E_1•type = bool else E_1•type = error;}

  E → true {E•type = bool}

  E → false {E•type = bool}

  E → num {E•type = int}

  $E_1 \rightarrow (E_2)$ {E_1•type = E_2•type}



Bottom-up Evaluation of SDT

Bottom-Up Evaluation of SDT

Semantic actions, associated with productions, are executed when a production is reduced

- The parser stack only knows grammar symbols (like E, T, id, etc).
- But for semantic actions (e.g., building a tree, calculating a value, printing code, translation), we need values or attributes.
- To carry out the semantic action, parser stack is extended with semantic stack.
- The set of actions performed on semantic stack are mirror reflections of parser stack.

SDT Example 5 the SDT Stack

- Consider the SDT:

  E → E + T { E.val = E_1.val + T.val }

  E → T { E.val = T.val }

  T → id { T.val = id.lval }

- Sample input: 2 + 3
  - id1.lval = 2
  - id2.lval = 3

Step	Parse Stack	Semantic Stack	Input	Action	Semantic Action
1	$ 0		id_1 + id_2 $		
2	$ 0 id 1	2	+ id_2 $	Shift id_1	Push id_1.lval = 2
3	$ 0 T 3	2	+ id_2 $	Reduce T → id	T.val = id.val = 2
4	$ 0 E 2	2	+ id_2 $	Reduce E → T	E.val = T.val = 2
5	$ 0 E 2 + 4	2	id_2 $	Shift +	No action
6	$ 0 E 2 + 4 id 5	2 3	$	Shift id_2	Push id_2.lval = 3
7	$ 0 E 2 + 4 T 6	2 3	$	Reduce T → id	T.val = id.val = 3
8	$ 0 E 2	5	$	Reduce E → E + T	E.val = 2 + 5 = 5
9	$ 0		$	Accept	Final result E.val = 5

SDT Example 6 to Print the Final Output

- Consider the following SDT:

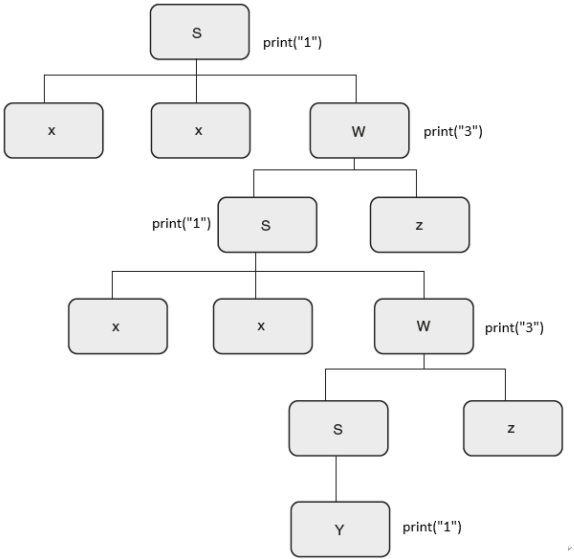
  S → xxW { print(“1”);}

  S → y { print(“2”);}

  W → Sz { print(“3”);}

- Input String: x^4yz^2

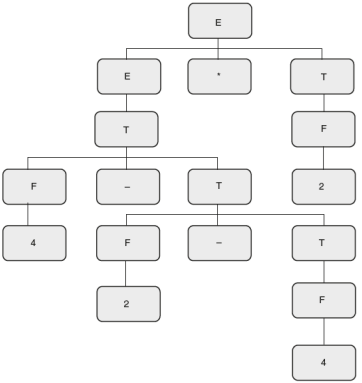
- Final output: 23131



SDT Example 7 To Count the total Reductions Performed

- Consider the SDT given below:

  E → E * T { E•val = E•val * T•val;}
  E → T { E•val = T•val;}
  T → F - T { T•val = F•val - T•val;}
  T → F { T•val = F•val;}
  F → 2 { F•val = 2;}
  F → 4 { F•val = 4;}



- Using the SDT construct parse tree and evaluate string 4 - 2 - 4 * 2
- It is also required to compute the total number of reductions performed to parse the given input string. Modify the SDT to find the number of reductions.

SDT Example 7 continued

- Hence, the SDT after modification:

E → E * T { E•val = E•val * T•val;
E•red = E•red + T•red + 1;}

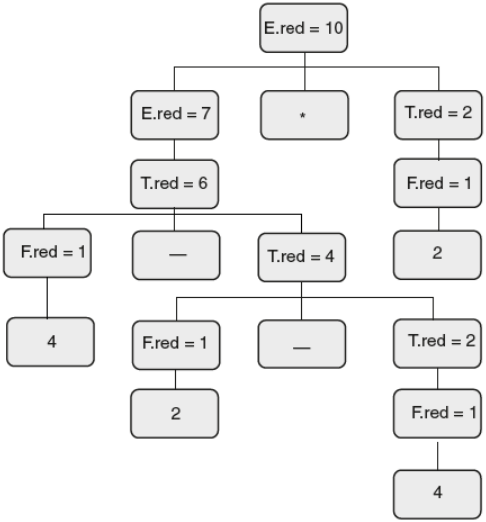
E → T { E•val = T•val; E•red =
T•red + 1;}

T → F – T { T•val = F•val – T•val;
T•red = F•red + T•red + 1;}

T → F { T•val = F•val; T•red =
F•red + 1;}

F → 2 { F•val = 2; F•red = 1;}

F → 4 { F•val = 4; F•red = 1;}

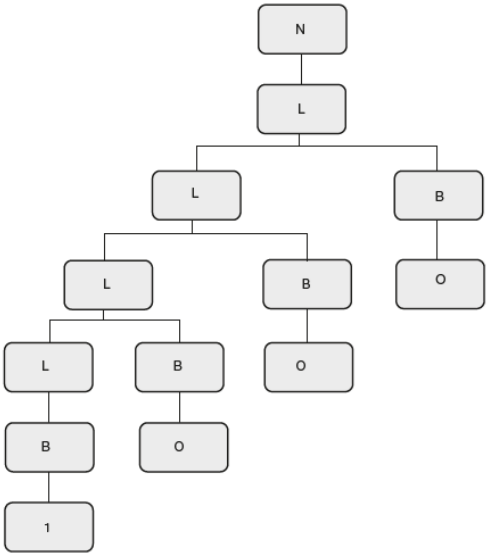


SDT Example8 Counting the digits in a Binary Number

- Write an SDT to count the number of binary digits. For example, “1000” is 4.
  - Consider the grammar:

    $N \rightarrow L$
    $L \rightarrow LB$
    $L \rightarrow B$
    $B \rightarrow 0$
    $B \rightarrow 1$

  - consider 1000 as an input
  - The parse tree will be:



SDT Example8 continued

- SDT will be:

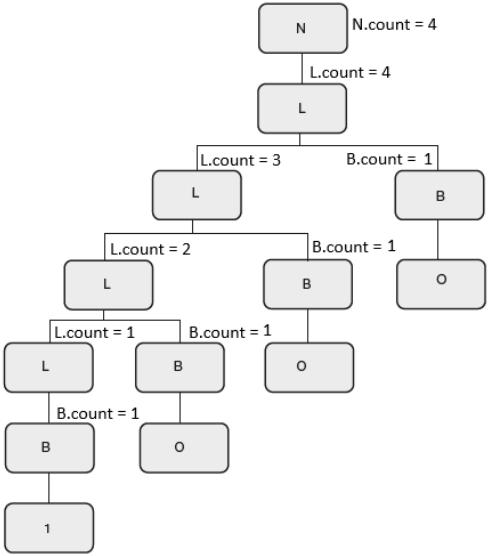
  N → L { N•count = L•count;}

  L → LB { L•count = L•count + B•count;}

  L → B { L•count = B•count;}

  B → 0 { B•count = 1;}

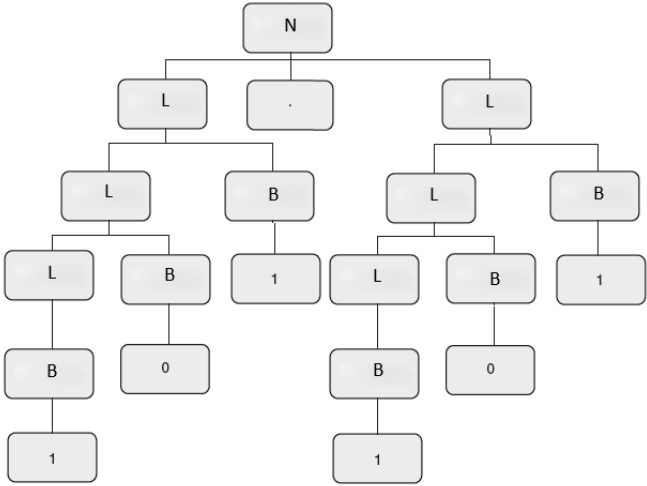
  B → 1 { B•count = 1;}



SDT Example 9 Binary to Decimal conversion

- Write an SDT to convert binary to decimal. For example, the binary number 101.101 denotes the decimal number 5.625.
  - Consider the grammar:

    $N \rightarrow L_1 \bullet L_2$
    $N \rightarrow L\ L \rightarrow LB$
    $L \rightarrow B$
    $B \rightarrow 0$
    $B \rightarrow 1$
  - consider 101.101 as an input
  - The corresponding parse tree will be



SDT Example 9 continued

SDT will be:

$$N \rightarrow L_1.L_2 \ \{N\bullet dval = L_1\bullet dval \ . \ L_2\bullet dval \ / \ 2^{L_2.nd};\}$$

- SDT for left sub tree:

  L → LB { L•dval = L•dval * 2 + B•dval;}

  L → B { L•dval = B•dval;}

  B → 0 { B•dval = 0;}

  B → 1 { B•dval = 1;}

- SDT for right sub tree:

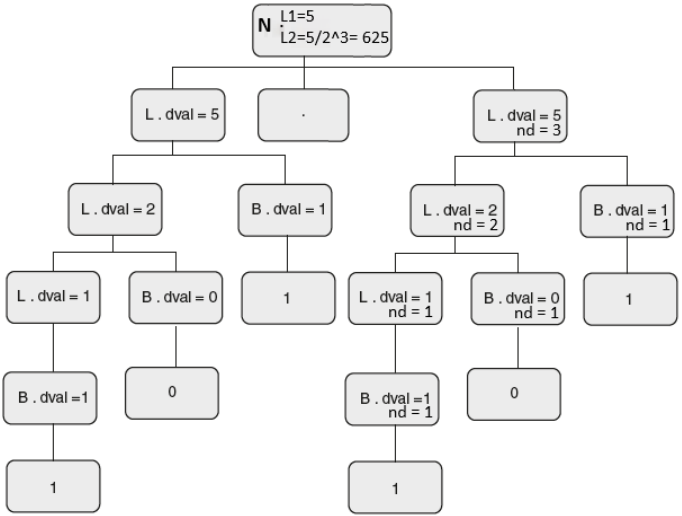
  L → LB { L•dval = L•dval * 2 + B•dval; L•nd = L•nd + B•nd;}

  L → B { L•dval = B•dval; L•nd = B•nd;}

  B → 0 { B•dval = 0; B•nd = 1; }

  B → 1 { B•dval = 1; B•nd = 1;}

SDT Example 9 Annotated Parse Tree



Parse Tree vs Syntax Tree

Parse Tree / Concrete Syntax Tree

It is a tree produced while checking the syntax of the programming language construct

- A parse tree gives the complete syntax of a string
- It tells you what the syntax behind the string is
- Hence it is called concrete syntax tree

Syntax Tree / Abstract Syntax Tree

A condensed form of a parse tree is called syntax tree

- It abstracts/hides away unnecessary terminals and nonterminals
- Hence, it is even called abstract syntax tree(AST).

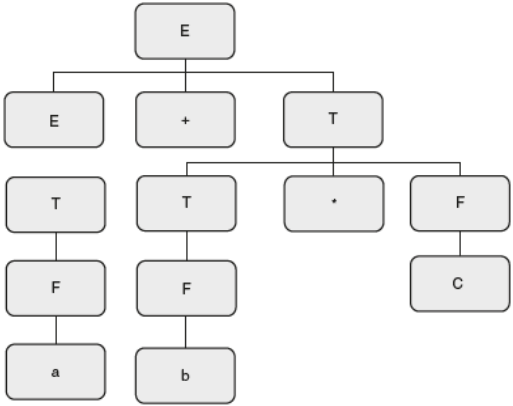
SDT for Creation of the Syntax Tree

- Consider the grammar:

  $E \rightarrow E + T$
  $E \rightarrow T$
  $T \rightarrow T * F$
  $T \rightarrow F$
  $F \rightarrow id$

- Assume input a + b * c

- Parse Tree:



SDT for Creation of the Syntax Tree continued

- Creation of a Syntax Tree:
  1. creation of a new node:
     mknode(left node, data, right node)
  2. pointer to a node: **nptr**

- Hence, the SDT will be:

  E → E + T { E•nptr = mknode (E•nptr, “+”, T•nptr);}
  E → T { E•nptr = T•nptr;}

  T → T * F { T•nptr = mknode (T•nptr, “*”, F•nptr);}

  T → F { T•nptr = F•nptr;}

  F → id { F•nptr = mknode (NULL, id•lvalue, NULL);}

- Syntax Tree: