Table of Contents

Class Fifteen - 08.10.2018

Keywords:

- Syntax Analysis

Role of Parser: The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language. For well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

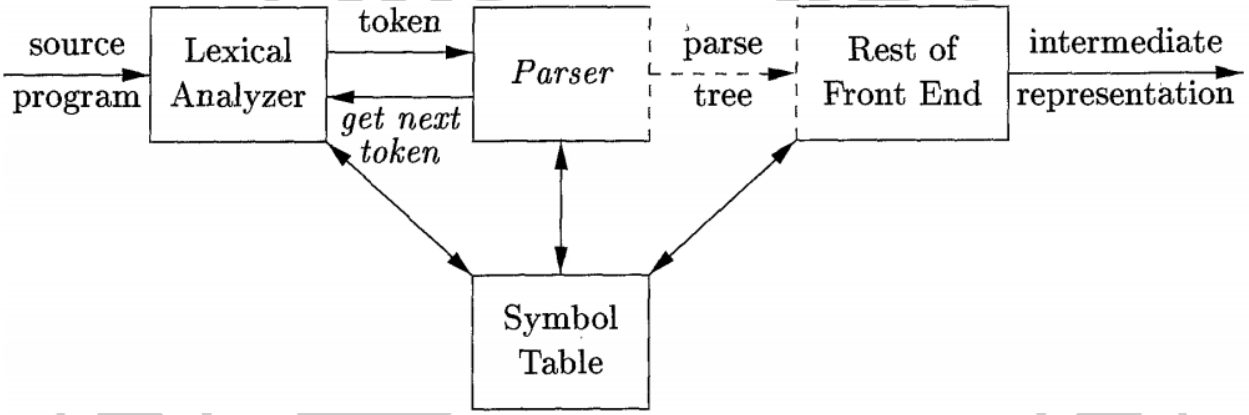


Figure: Position of parser in compiler model

The parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. There are three general types of parsers for grammars. Such as:

- Universal
- Top-down
- Bottom-up

Universal Parsing: Universal Parsing methods can be the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. These general methods are, however, too inefficient to use in production compilers.

Cocke-Younger-Kasami Algorithm: The Cocke–Younger–Kasami algorithm (CYK) is a parsing algorithm for context-free grammars. It employs bottom-up parsing and dynamic programming. The standard version of CKY can only recognize languages defined by context-free grammars in Chomsky Normal Form (CNF). It uses the grammar directly. The main algorithm is:

Step-1: Considers every possible consecutive subsequence of letters and sets K ∈ T[i,j] if the sequence of letters starting from i to j can be generated from the non-terminal K.
Step-2: Once it has considered sequences of length 1, it goes on to sequences of length 2, and so on.
Step-3: For subsequences of length 2 and greater, it considers every possible partition of the subsequence into two halves, and checks to see if there is some production A -> BC such that B matches the first half and C matches the second half. If so, it records A as matching the whole subsequence.
Step-4: Once this process is completed, the sentence is recognized by the grammar if the entire string is matched by the start symbol.

Earley's Algorithm: The Earley parser is an algorithm for parsing strings that belong to a given context-free language, though it may suffer problems with certain nullable grammars. The algorithm is a chart parser that uses dynamic programming; it is mainly used for parsing in computational linguistics. The main algorithm is:
Step-1: $S_0 = \{[S \rightarrow \bullet P (0)]\}$
Step-2: For $0 \leq i \leq n$ do:
Process each item $s \in S_i$ in order by applying to it a single applicable operation among:

(a) Predictor (adds new items to S_i)
(b) Completer (adds new items to S_i)
(c) Scanner (adds new items to S_{i+1})

Step-3: If $S_{i+1} = \emptyset$ Reject the input.
Step-4: If i = n and $[S \rightarrow P\bullet (0)] \in Sn$ then Accept the input.

Top-down: Top-down parsing is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. Top-down methods build parse trees from the top to the bottom. Example: LL parsers are a type of parser that uses a top-down parsing strategy.

Bottom-up: Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Example: LR parsers are a type of parser that uses a bottom-up parsing strategy.

Representatives Grammar: This grammar does not enforce precedence and it does not specify left vs right associativity. Associativity and precedence are captured in this grammar. It can't be used for top-down parsing because it is left recursive.
In here, E represents expressions consisting of terms separated by + signs, T represents terms consisting of factors separated by * signs, and F represents factors that can be either parenthesized expressions or identifiers:

E → E+T
T → T * F | F
F →(E) | id

Expression of above grammar belongs to the class of LR grammars that are suitable for bottom-up parsing. But it cannot be used for top-down parsing because it is left recursive. Therefore for top-down parsing, the following non-left-recursive variant of the expression grammar will be used:

E → TE'
E'→ +TE' | ϵ
T → FT
T' → *FT'| ϵ
F →(E) | id

Class Sixteen - 11.10.2018

Keywords:

- Role of the parser
- Benefits of grammar
- Error handling / Error challenges / Error recovery

Benefits of Grammar:

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
- Reveal syntactic ambiguities.
- A grammar is useful for translating source program into correct object code and for detecting errors.
- A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks.

Error-Handling Strategies:

- Lexical errors include misspellings of identifiers, keywords, or operators e.g., the use of an identifier elipseSize instead of ellipseSize.
- Syntactic errors include misplaced semicolons or extra or missing braces; that is, "{" or "}."
- Semantic errors include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.
- Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.

Error-Recovery Strategies:

- **Panic-Mode Recovery:** When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

- **Phrase-Level Recovery:** When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

- **Error Productions:** Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

- **Global Correction:** The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.

The error handler in a parser has goals that are simple but challenging. Namely-

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

Notational Conventions: To avoid always having to state that "these are the terminals," "these are the nonterminals," and so on, the following notational conventions for grammars are used to remind.

- Terminal symbols are :
  (a) Lowercase letters early in the alphabet, such as a, b, c.
  (b) Operator symbols such as +, *, and so on.
  (c) Punctuation symbols such as parentheses, comma, and so on.
  (d) The digits 0, 1, 2, . . , 9.
  (e) Boldface strings such as id or if

- Non-Terminal symbols are :
  (a) Uppercase letters early in the alphabet, such as A, B, C.
  (b) The letter S, which, when it appears, is usually the start symbol.
  (c) Lowercase, italic names such as expr or stmt.
  (d) uppercase letters may be used to represent nonterminals for the constructs.

Class Seventeen - 14.10.2018

Keywords:

- Elimination of left recursive

Elimination of left recursive:

- A grammar is left recursive if it has a nonterminal A such that there is a derivation A =+= Aa for some string a.
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.
- In simple left recursion there was one production of the form A -> Aa.

Left Recursive: E → E+T/T

Non-Left Recursive: A → βA'

A → Aα β

Conversion:

T→T * F|F

T→FT'

T→+FT'| ϵ

Examples:

1. C → Caab|d
   C → dC'
   C' → aabc'| ϵ
2. A→Aaab | BC
   A→ BCA'
   A→ aabA'| ϵ
3. B→Bc|d
   B→dB'
   B→CB'| ϵ
4. S→Aa|b
   A→Sc|d
   S→Aa|d
   S→SCa|d
   S→bds'
   S→acs'| ϵ
5. L→Ls|S
   L→SL'
   L'→SL'| ϵ
6. S→SOSIS | 01
   S→01S'
   S'→0SISS'| ϵ
7. A→A+A/B
   A→BA'
   A→AA'| ϵ
8. A→Abc|Acd|CD|XY
   A→ CDA'|XYA'
   A→bcA'|cdA'| ϵ
9. A→Ac|Aab|bc|c
   A→bcA'| CA'

Class Eighteen - 15.10.2018

Keywords:

- Left factoring

Left factoring: Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that sometimes it is not clear which of two alternative productions to use to expand a nonterminal A. We may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

Formula of Left Factoring:

- A → $\alpha\beta_1$ | $\alpha\beta_2$ are two A-productions.
- The input begins with a nonempty string derived from α.
- We do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.
- However, we may defer the decision by expanding A to αA'.
- Then, after seeing the input derived from α we expand A' to β_1 or β_2.
- Left-factored the original productions become, A→αA

  A'→ β_1 | β_2

Class Nineteen - 18.10.2018

Keywords:

- Left factoring
- Top-down parse tree

Left Factoring Method:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

- For each nonterminal A find the longest prefix α common to two or more of its alternatives.
- If $\alpha \neq \epsilon$ (there is a nontrivial common prefix), we have to replace all the A productions A —> $\alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots \mid \alpha\beta_n\, \gamma$ where γ represents all alternatives that do not begin with α by

  $A \rightarrow \alpha A' \mid \gamma$

  A —> $\beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$

  Where A' is a new nonterminal.
- Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Common Prefix Problem

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$

Example:

1. S→iEtS| iEtSeS|a
   S'→eS| **ϵ**
   E→b
2. A→aAB|aA
   A→aAA'
   A'→ B| ϵ
3. T→T+T|T
   T→T T'
   T'→ +T T|ϵ
4. S→bSSaaS|bSSaSb|bSb|aϵ
   S→bSS'|a
   S'→SaaS| SaSb|b
   S'→SaS"|b
   S"→as|Sb

Top down parsing according to grammar (non-left recursive)

E→ TE'

E'→+TE'|ϵ

T→FT'

T'→FT'|ϵ

F→(E)|id

The parsing tree is drawn below:

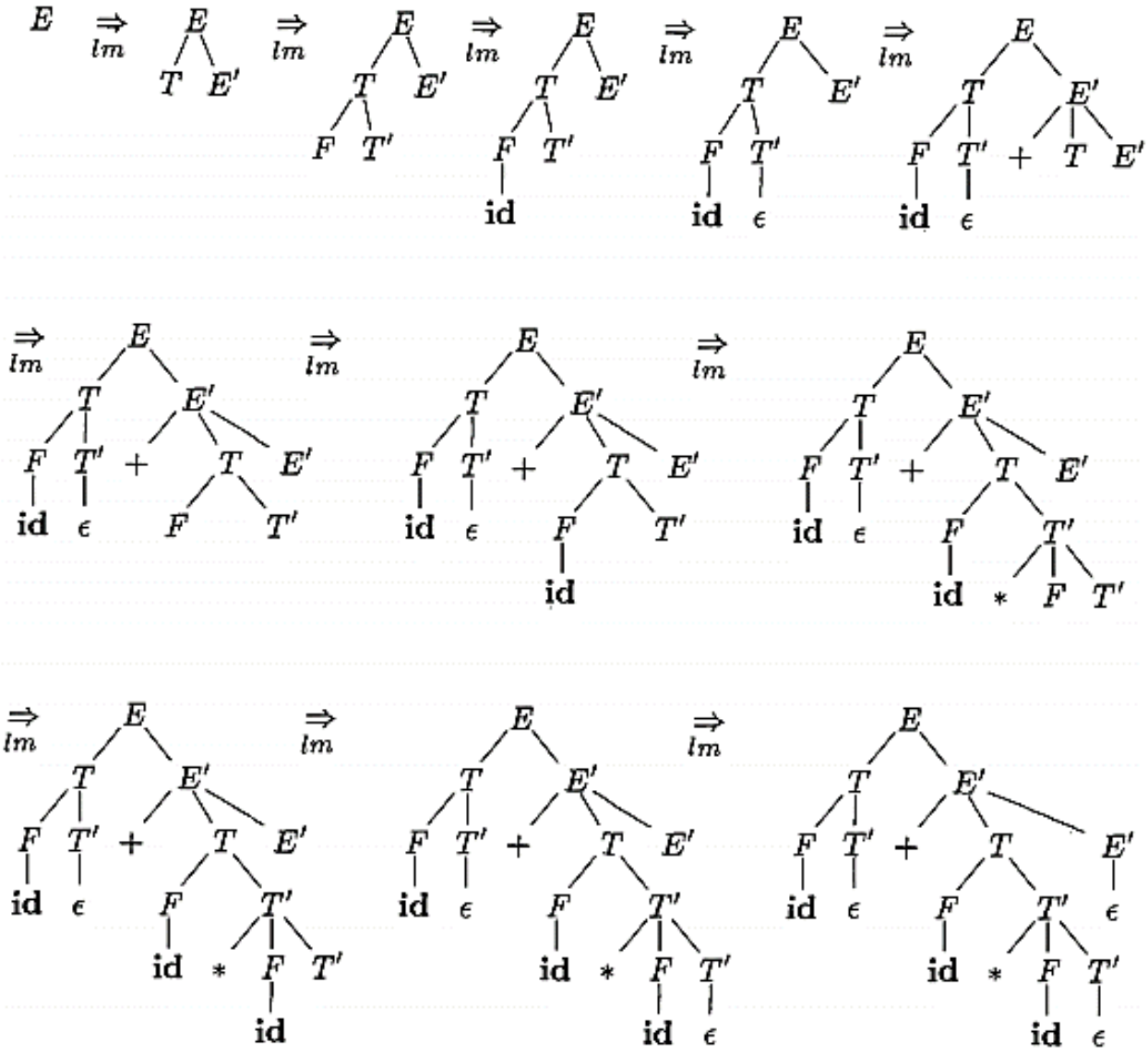


Figure: Top down parse for id+id*id

First and Follow: The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

First: For any string α of grammar symbols, we define FIRST(α) to be the set of terminals that occur as the first symbol in a string derived from α. So, if α⇒*cβ for c a terminal and β a string, then c is in FIRST(α). In addition, if α⇒*ε, then ε is in FIRST(α).

Follow: For any nonterminal A, FOLLOW(A) is the set of terminals x, that can appear immediately to the right of A in a sentential form. Formally, it is the set of terminals c, such that S⇒*αAcβ. In addition, if A can be the rightmost symbol in a sentential form (i.e., if S⇒*αA), the endmarker $ is in FOLLOW (A).

Example:

1. Find first and follow for the grammars below:

Grammar	First	Follow
S→ABCDE	{a, b, c, d, e}	{$}
A→a|ϵ	{a, ϵ}	{b, c}
B→ b|ϵ	{b, ϵ}	{c}
C→c	{c}	{d, e, $}
D→d|ϵ	{d, ϵ}	{ ϵ, 4}
E→ e|ϵ	{e, ϵ}	{$}

2. Find first and follow for the grammars below:

Grammar	First	Follow
S→Bd|Cb	{ a, d, c, b }	{$}
B→aB|ϵ	{ a, ϵ }	{d}
C→cC |ϵ	{ c, ϵ }	{b}

3. Find first and follow for the grammars below:

Grammar	First	Follow
S→aABbc	{a}	{$}
A→c|ϵ	{ c, ϵ }	{d, b}
B→ d|ϵ	{ d, ϵ }	{b}

Class Twenty - 21.10.2018

Keywords:

- First and Follow
- Parsing Table

First and follow for the grammars below:

Grammar	First	Follow
E→TE'	{ id, (}	{ $,) }
E'→+TE'|ϵ	{ +, ϵ }	{ $,) }
T→FT'	{ id, (}	{ +, $ }
T'→*FT'| ϵ	{ *, ϵ }	{ +, $}
F→id| (E)	{ id, ϵ }	{*, +, $ }

Parsing Table for the above grammar is drawn below:

Non Terminal	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ ϵ	T'→*FT'		T'→ ϵ	T'→ ϵ
F	F→id			F→(E)		

Class Twenty One - 22.10.2018

Keywords:

- Class Test
- Parse the parsing table for specific string by particular grammar.

The parsing table for specific string by following grammar is:

E→ TE'
E'→+TE'|ϵ
T→FT'
T'→FT'|ϵ
F→(E)|id

Stack	Input	Action
$E	id+id*id$	
$T E’	id+id*id$	E→ TE’
$E’ T’ F	id+id*id$	T→FT’
$E’ T’ id	id+id*id$	F→id
$E’T’	+id*id$	
$E’	+id*id$	T’→ϵ
$E’T+	+id*id$	E’→+TE’
$E’T	id*id$	
$E’T’F	id*id$	T→FT’
$E’T’ id	id*id$	F→id
$E’T’	*id$	
$E’T’F*	*id$	T’→*FT’
$E’T’F	id$	
$E’T’id	id$	F→id
$E’T’	$	
$E’	$	T’→ϵ
$	$	E’→ϵ

Table: Moves made by a predictive parser on input id+id*id

If stack ad input is up to last then the string id+id*id will be accepted by particular grammar based on parsing table or not.

Class Twenty Two- 25.10.2018

Keywords:

- Parse Tree
- Bottom up parse tree
- Derivation – LMD and RMD
- Ambiguity.

Parse Tree: A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production

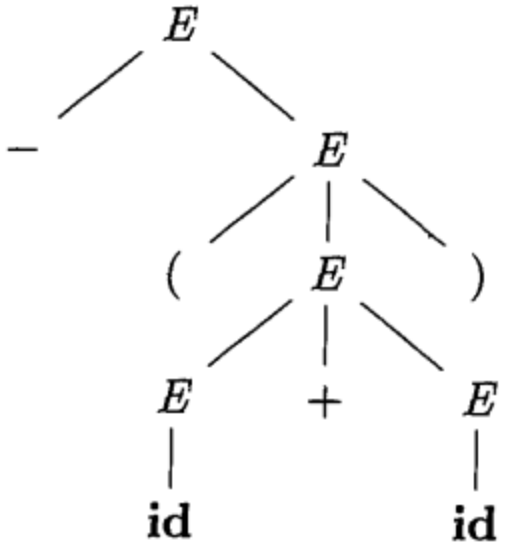


Figure: Parse tree for – (id+id)

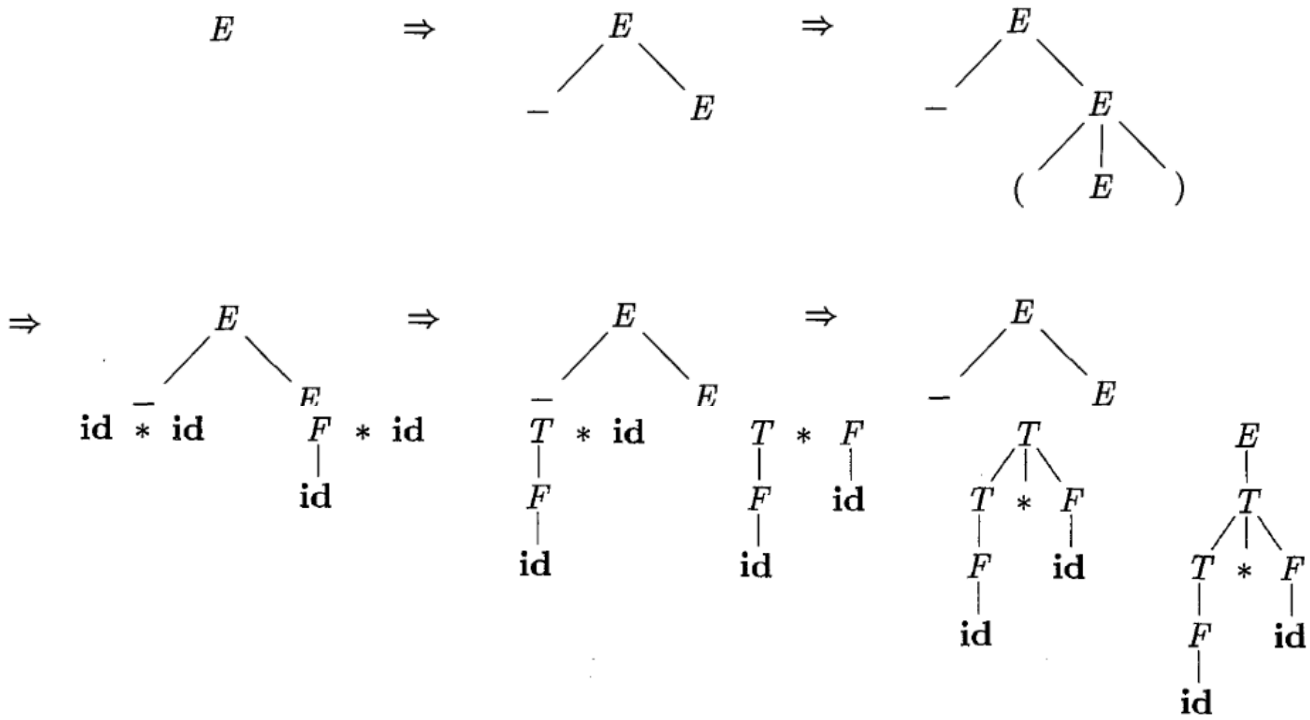


Figure: Sequence of parse tree for derivation

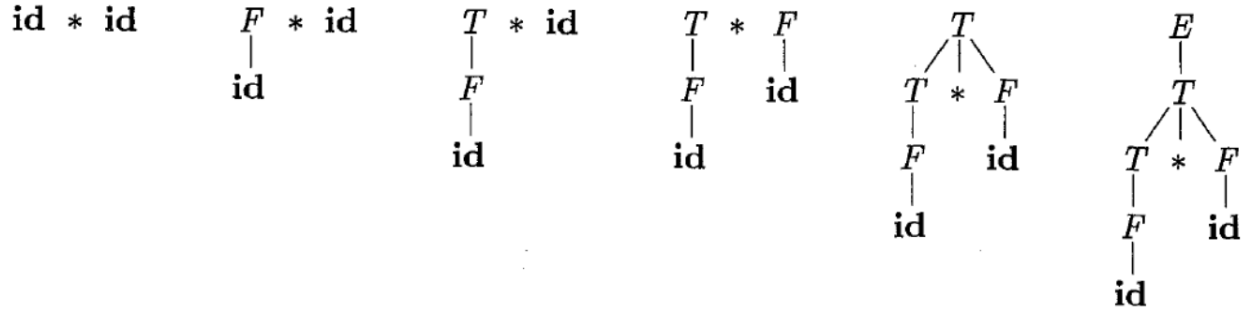


Figure: Bottom-up parse for id*id

Ambiguity: A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Example: The arithmetic expression grammar permits two distinct leftmost derivations for the sentence id + id * id.

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

Parse tree for id+id*id drawn below:

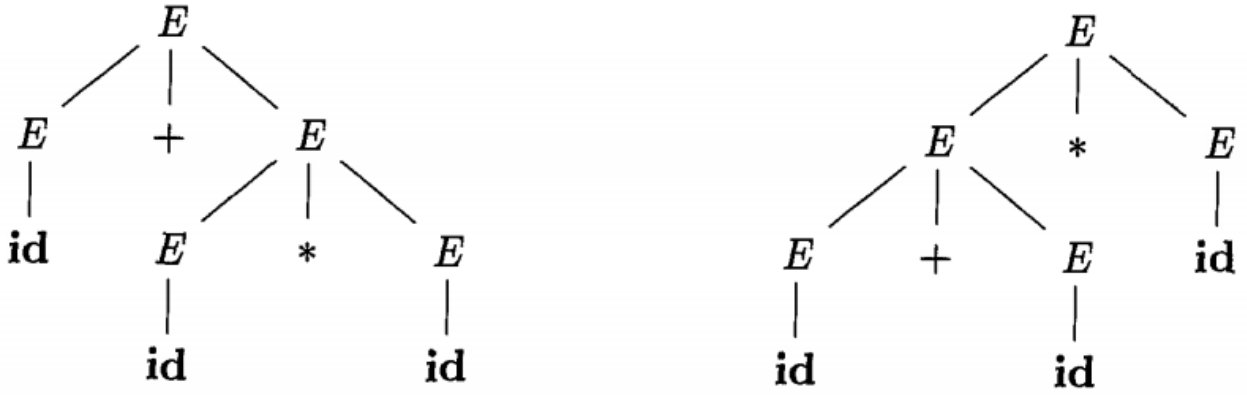
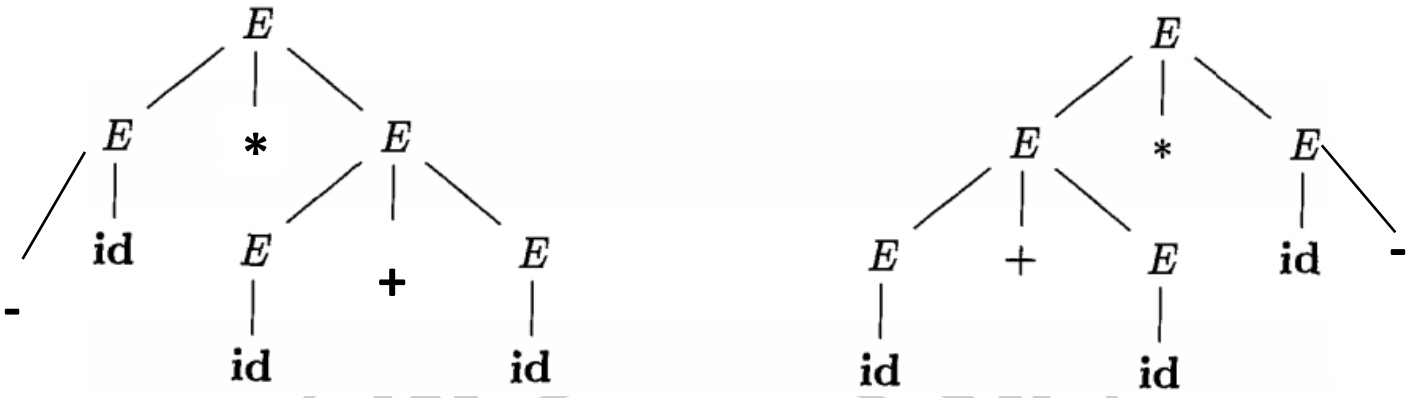


Figure: Two parse tree for id+id*id

Example: if the input is –(id*(id+id)) then find out the LMD.



Or else
-(E)
=-(E*E)
= [id*(E+E)]
=-[id* (id+id)]
=-id* (id+id)

Class Twenty Three- 28.10.2018

Keywords:

- Assignment Review
- LMD & RMD
- CFG

Leftmost Derivation (LMD): In leftmost derivation, at each and every step the leftmost non-terminal is expanded by substituting its corresponding production to derive a string.

Rightmost Derivation (RMD): In rightmost derivation, at each and every step the rightmost non-terminal is expanded by substituting its corresponding production to derive a string.

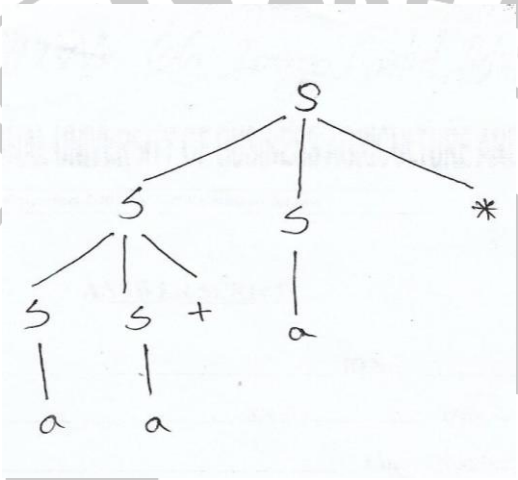
Example-1: Consider the following grammar S→SS+|SS*|a with the string aa+a* to answer the questions below:

a. Give a LMD of for the string
b. Give a RMD of for the string

c. Give a parse tree for the string
d. If the G ambiguous or unambiguous?

Answer:

a. S→ LM→SS+|SS*|a
   S → SS* → SS+S*→aS+S*→aa+S*→aa+a*
b. S→ RM→SS*→SS+S*→SS+a*→Sa+a*→aa+*
c. Parse Tree for aa+a*



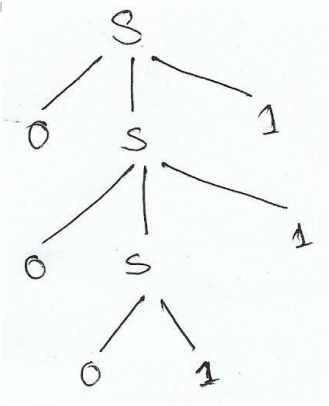
d. Unambiguous.

Example-2: Consider the following grammar S→OS1| 01 with the string 000111 to answer the questions below:

a. Give a LMD of for the string
b. Give a RMD of for the string
c. Give a parse tree for the string
d. If the G ambiguous or unambiguous?

Answer:

a. S→LM→0S1→00SS1→000111
b. S→RM→0S1→00S11→000111
c. Parse Tree for 000111
d. Unambiguous.

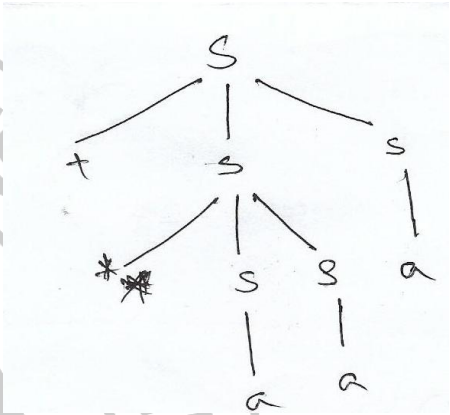


Example-3: Consider the following grammar S→+SS|*SS|a with the string +*aaa to answer the questions below:

a. Give a LMD of for the string
b. Give a RMD of for the string
c. Give a parse tree for the string
d. If the G ambiguous or unambiguous?

Answer:

a. S→LM→+SS→+SS→+*SSS→+*aSS→+*aaS→+*aaa
b. S→RM→+SS→+Sa→+*SSa→+*Saa→+*aaa
c. Parse Tree for+*aaa
d. Unambiguous



Example-4: Consider the following grammar !S→S(S)S | ϵ with the string (()())to answer the questions below:

a. Give a LMD of for the string
b. Give a RMD of for the string

Answer:

a. S→LM→S(S)S→(S)S →(S(S)S)S→((S)S)S→(()S)S→(()S(S)S)S
→(()(S)S)S→(()()S)S→(()())S→(()())
b. S→RM→S(S)S→(S)S →S(S(S)S)→S(S(S))→S(S())→S(S(S)S())
→S(S(S)())→S(S()())→S(()())→(()())

Example-5: Consider the following grammar !S→S+S |SS|(S)|S*|a with the string (a+a)*a to answer the questions below:

a. Give a LMD of for the string
b. Give a RMD of for the string

Answer:

a. S→LM→SS→S*S→(S)*S→(S+S)*S→(a+S)*S→(a+a)*S→(a+a)a
b. S→RM→SS→Sa→ S*a→(S)*a→(S+S)*a→(S+a)*a→(a+a)*a

Example-6: Consider the following grammar !S→(L)|a with the string ((aa)a(a)) to answer the questions below:

a. Give a LMD of for the string
b. Give a RMD of for the string

Answer:

a. S→LM→(L)→(L,S) →(L,S,S) →((S),S,S) →((L)S,S) →((L,S),S,S) →((S,S),S,S) →((a,S),S,S)→((a,a),S,S)→((a,a),a,S)→((a,a),a(L)) →((a,a),a,(S))→((a,a),a,a)
b. S→RM→(L)→(L,S) →(L,(L)) →(L,(a)) →(L,S(a)) →(L,a(a)) →(S,a,(a)) →((L),a(a)) →((L,S),a(a)) →((S,S),a(a)) →((S,a),a(a)) →((a,a),a,(a))

Class Twenty Four - 29.10.2018

Keywords:

- CFG
- LL(1)

Context Free Grammar: That is used to specify the syntax of a language. A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java.

A context-free grammar has four components:

1 A set of terminal symbols, sometimes referred to as \tokens." The terminals are the elementary symbols of the language defined by the grammar.
2 A set of nonterminals, sometimes called \syntactic variables." Each nonterminal represents a set of strings of terminals.
3 A set of productions where each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminals, called the body or right side of the production.
4 A designation of one of the nonterminals as the start symbol.

LL(1) Grammar: Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1).

1. The first *L* stands for scanning the input from **l**eft to right,
2. The second *L* stands for producing a leftmost derivation,
3. The 1 stands for using one input symbol of lookahead at each step to make parsing action decision.

A grammar G is LL(1) if and only if whenever A —> α | β are two distinct productions of G the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \Rightarrow \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise,$\alpha \Rightarrow \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

Algorithm for Construction of a Predictive Parsing Table:
Input: grammar g.
Output: parsing table m.
Method: For each production A —> α of the grammar. do the following:

1. For each terminal a in FIRST(A). add A to M[A. a] .
2. If ϵ, is in FIRST((α). then for each terminal b in FOLLOW(A). add A→ α to M[A. b].
   If ϵ is in FIRST(α) and $ is in FOLLOW(A), add A —> α M[A. $] as well.

LL(1) Parser
Input buffer: our string to be parsed. We will assume that s end is marked with a special symbol S.
Output: a production rule representing a step of the derivation sequence (left-most derivation) of the sting in the input buffer.
Stack: contains the grammar symbols. At the bottom of the stack, there is a special end marker symbol $. Initially the stack contains only the symbol $ and the starting symbol S. $S E initial stack when the stack is emptied, the parsing is completed.
Parsing table: a two-dimensional array M[A,a]. Each row is a non-terminal symbol. Each column is a terminal symbol or the special symbol $. Each entry holds a production rule.

Table-Driven Predictive Parsing:
Input: A string w and a parsing table M for grammar G.
Output: If w is in L(G), a leftmost derivation of w; otherwise, an error indication.

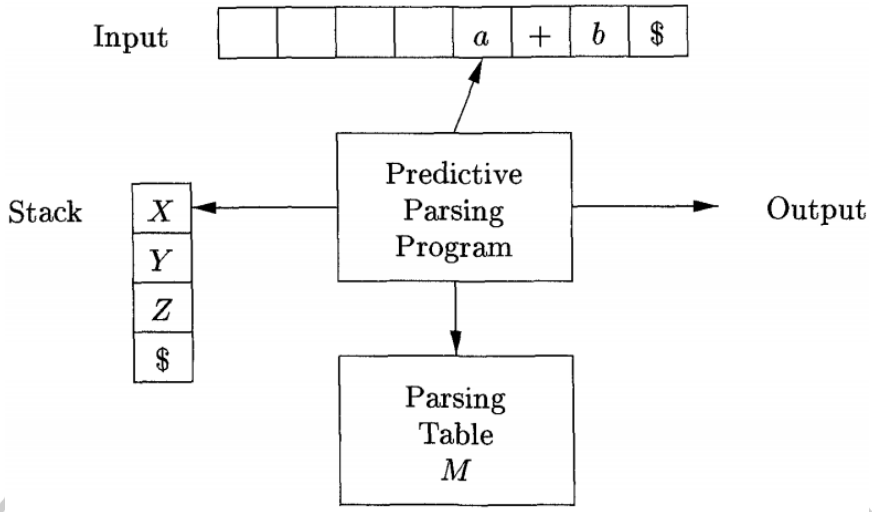


Figure: Model of Table-Driven Predictive Parser

Method: Initially, the parser is in a configuration with w$ in the input buffer and the start symbol S of G on top of the stack, above S. The program uses the predictive parsing table M to produce a predictive parse for the input.

Class Twenty Five - 01.11.2018

Keywords:

- Chapter-5
- Syntax Directed Translation

Syntax Directed Definition: Syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. Example:

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

In SDD there are two types of attributes

Synthesized Attribute: A synthesized attribute for a nonterminal A at a parse-tree. It can take values from child. It is also called as S attribute. Terminal can have synthesized value.

Inherited Attribute: An inherited attribute for a nonterminal B at a parse-tree An inherited attribute can take value form parents and can help siblings but not from child. It could never have any terminal.

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Figure: SDD for Desk calendar

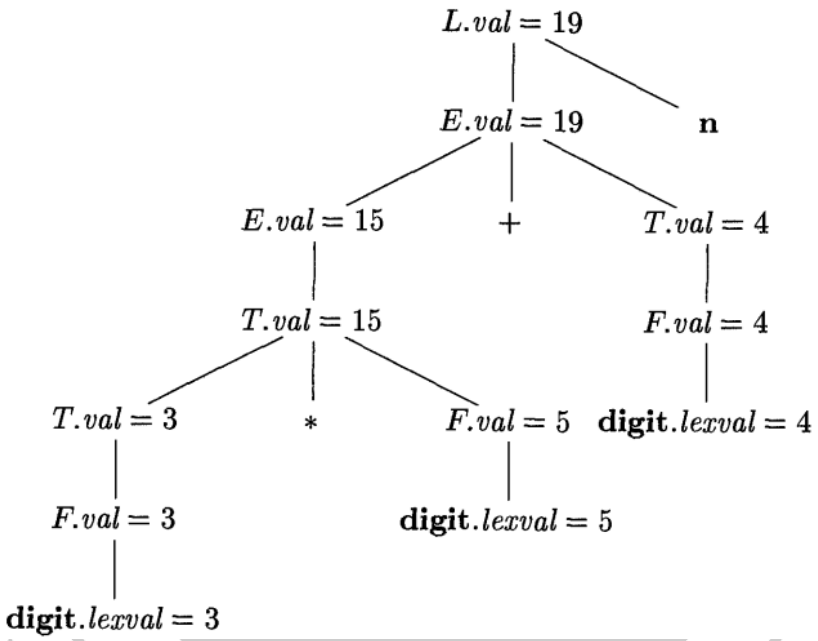


Figure: Annotated Parse Tree for 3 * 5 + 4n

Class Twenty Six - 04.11.2018

Keywords:

- SDT / SDD

Draw annotated parse tree for 2 + 3 *4

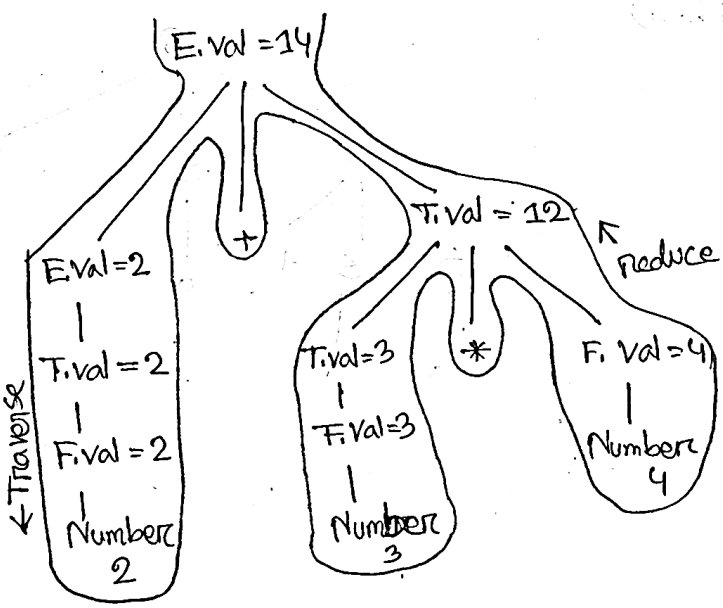


Figure: Annotated parse tree for 2 + 3 *4

SDT	**Action**
S→XXW	{printf $(1)_i$} y→(2)
/y	{printf $(2)_i$} Sz→(3)
w→SZ	{printf $(3)_i$} XXXW→(1)
	and SZ (3)

for string XXXXYZZ

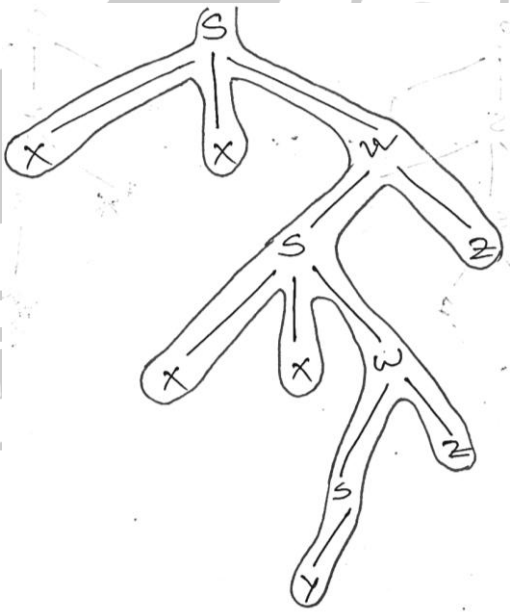


Figure: Annotated parse tree for XXXXYZZ

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Table: An SDD based on a grammar suitable for top-down parsing

Class Twenty Seven - 05.11.2018

Keywords:

- SDT / SDD
- Review class and so on

Draw annotated parse tree for 1 * 2 * 3 (4+5)n

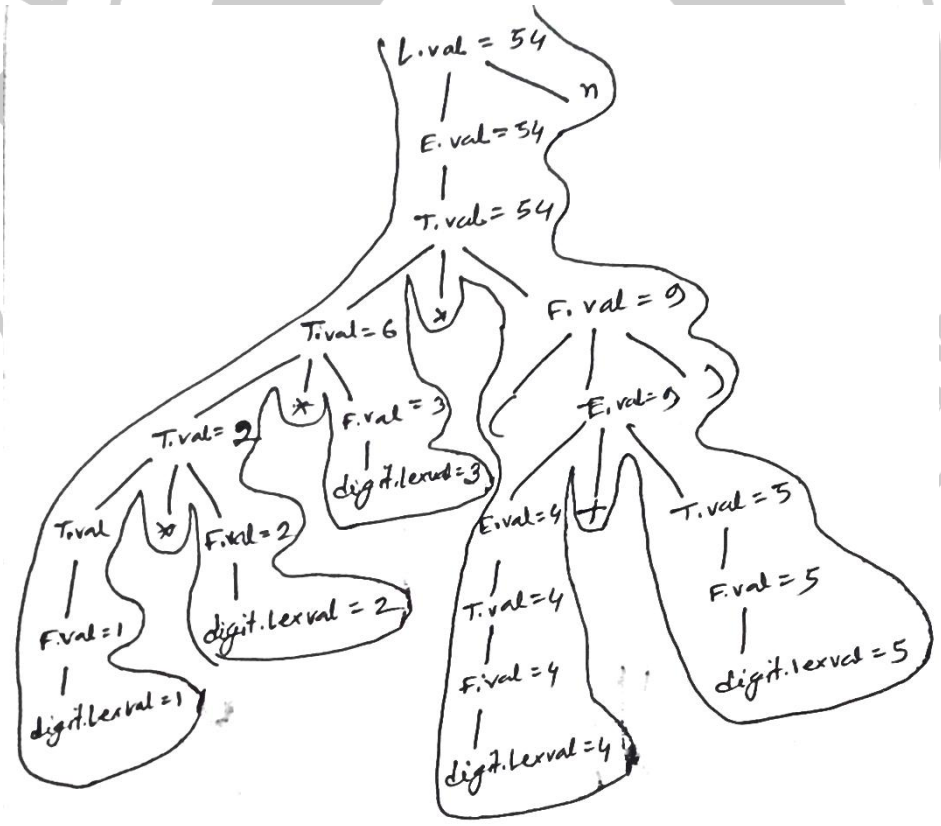


Figure: Annotated parse tree for 1 * 2 * 3 (4+5)n

Draw annotated parse tree for (3 +4) * (5+6)n

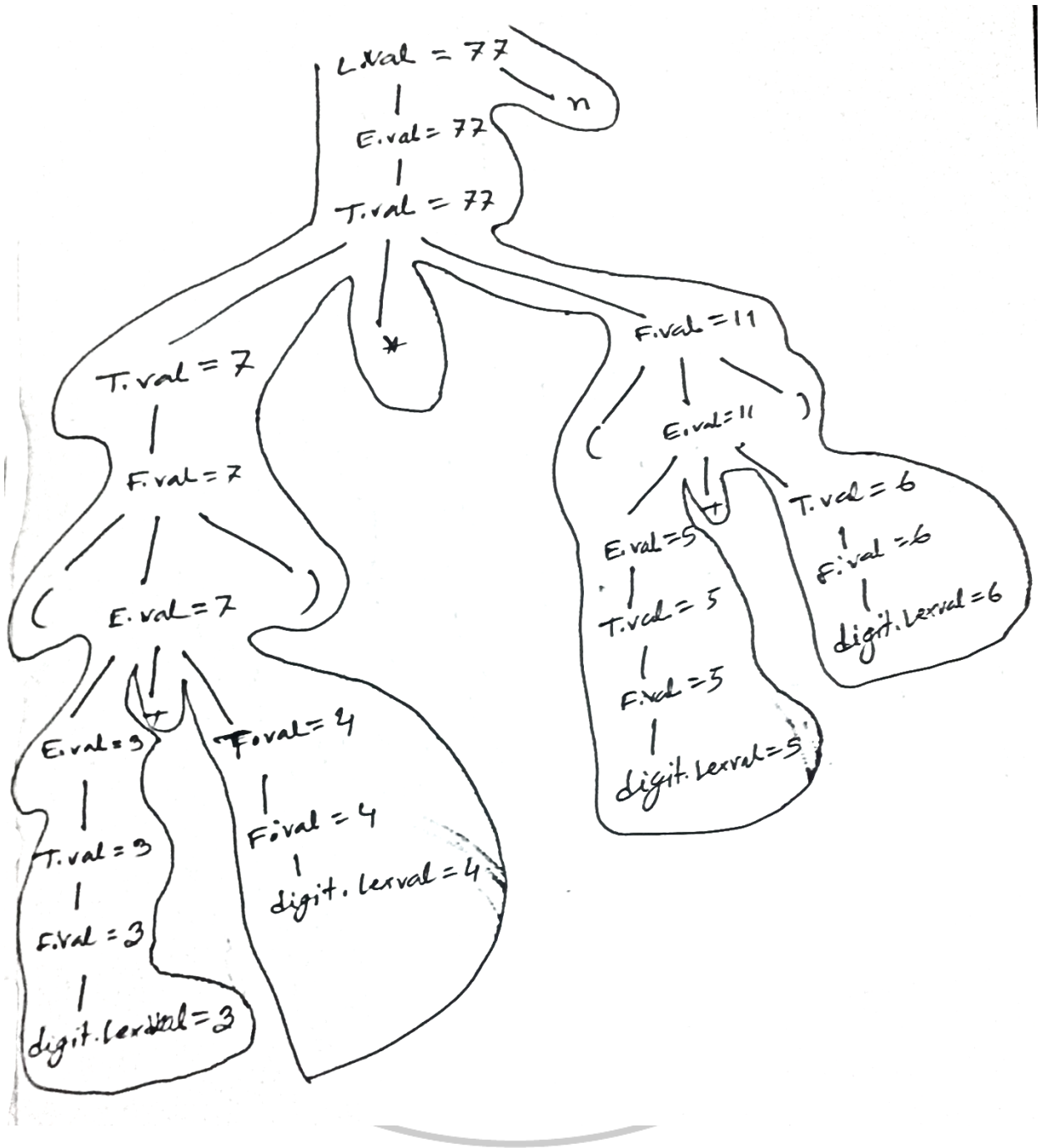


Figure: Annotated parse tree for (3 +4) * (5+6)n