# Course: Compiler Construction

# Week 6: Top-Down Parsing

**Lecturer:** Martha Gichuki

# Course Description

- The course begins with a coverage of basic Principles in Compiler Design with an introduction to Formal programming language translation and compiler design concepts; compilers, translators and interpreters.
- A coverage of Language theory, Parsing Context-Free Languages (Top - down and bottom - up parsing), translation specifications and machine- independent code optimization will then follow.

# Course Description …

- The main phases of compilation i.e. Lexical, Syntax and Semantic analysis will be taught.
- Code generation and Optimization, Symbol table design, Program compilation, Loading and execution will then be covered.
- Compilation techniques, Optimization, Design of a simple complete compiler. will culminate the course.

**Learning outcomes Week 6: Top-Down Parsing**

At the end of the lecture, you will be able to:

(i) Define Recursive Descent Parsing and Predictive Parsing

(ii)Describe backtracking technique

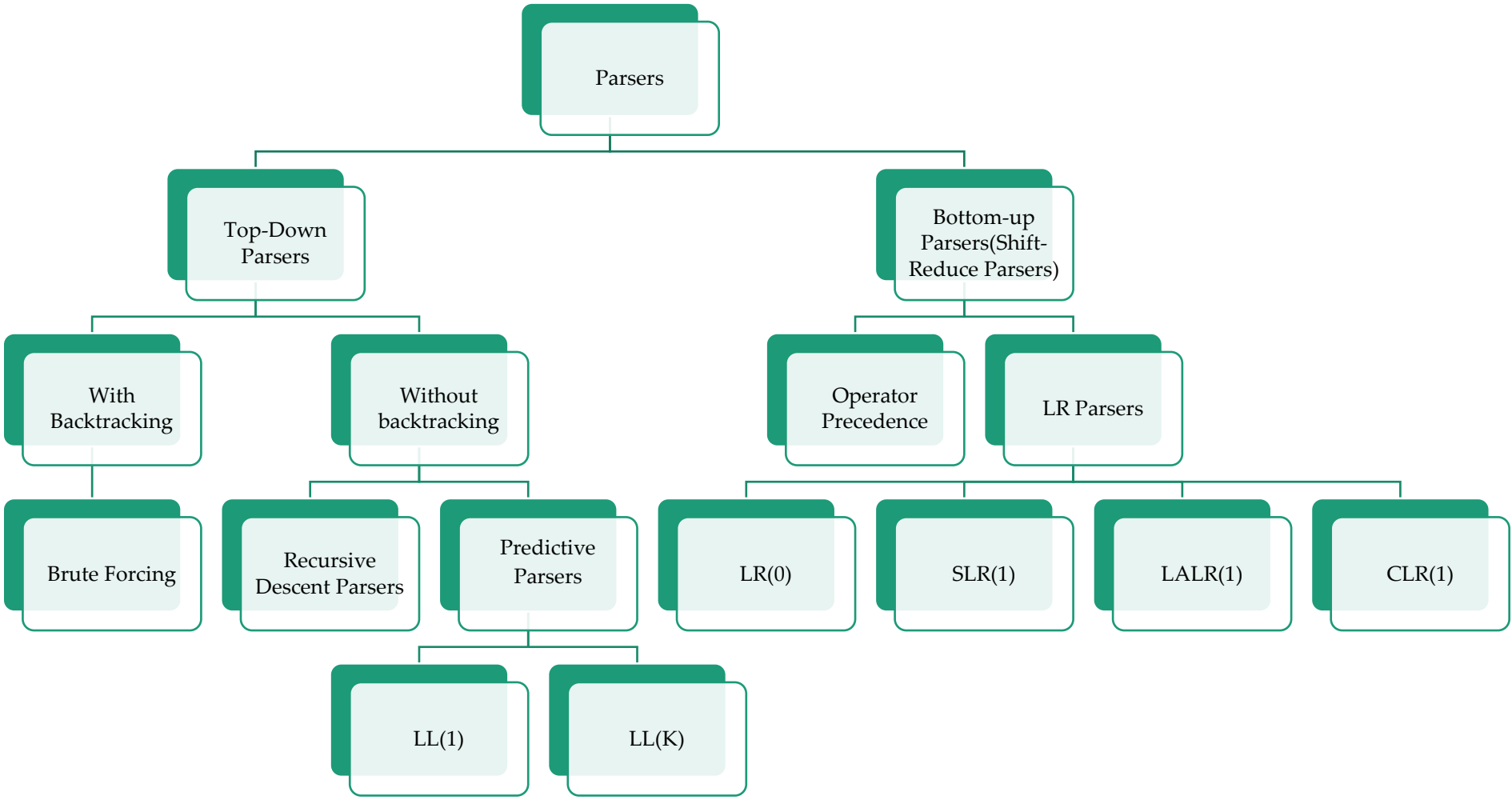
(iii) Describe lookahead set of terminals

# Definition: Parsing

- ✓**Parsing** is a convenient way to see how strings are derived from the start symbol.
- ✓It is a way of analyzing a string or text into **logical syntactic components**
- ✓Tokens are divided into parts and the relations between the parts can then be described[1]

1. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 86

# Classification of parsers[2]




2. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 324

# Top-Down Parsing

This parsing strategy finds the derivation of the input string **from the start symbol** of the grammar[3].

There are two main approaches for Top-Down parsing:

a) Recursive Descent Parsing

b) Predictive Parsing

3. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 86

# A. Recursive Descent Parsing[4]

- ✓ It is a common form of top-down parsing that uses a parsing technique, which recursively parses the input to make a parse tree from the top and the input is read from left to right.
- ✓ It is called recursive, as it uses recursive procedures to process the input.
- ✓ The main idea is that **each non-terminal** in the grammar is **implemented by a procedure** in the program.

4. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 242

# Recursive Descent Parsing …

- ✓ Generally, these parsers consist of a set of mutually recursive routines that may require back-tracking to create the parse tree.
- ✓ **Definition - Backtracking:** If one derivation of a production fails, the syntax analyzer restarts the process using **different rules** of the same production[5].
- ✓ Backtracking technique may process the input string more than once to determine the right production.

5. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 244

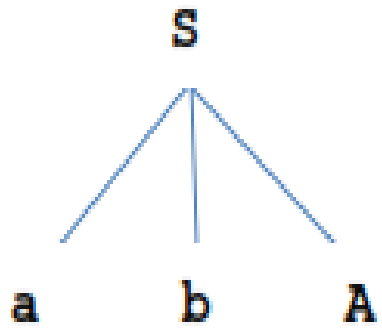
**Example:** Consider the grammar:

$S \rightarrow abA$

$A \rightarrow cd \mid c \mid e$

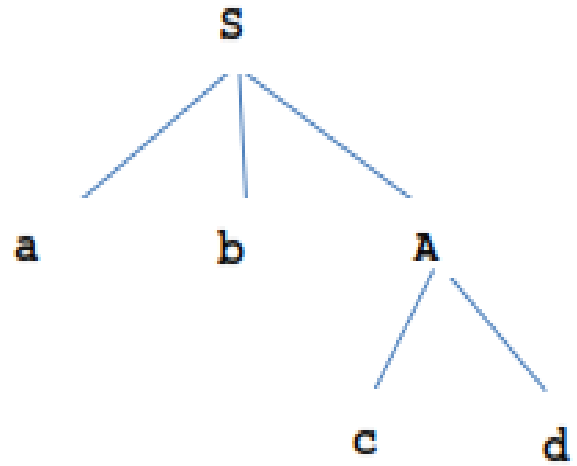
- ✓ For the input stream $ab$, the recursive descent parser starts by constructing a parse tree representing $S \rightarrow abA$ as shown below[6]:

6. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 244




$S \rightarrow abA$
$A \rightarrow cd \mid c \mid e$

- ✓ The stream is then expanded with the production $\boldsymbol{A \rightarrow cd}$ as shown below[7]

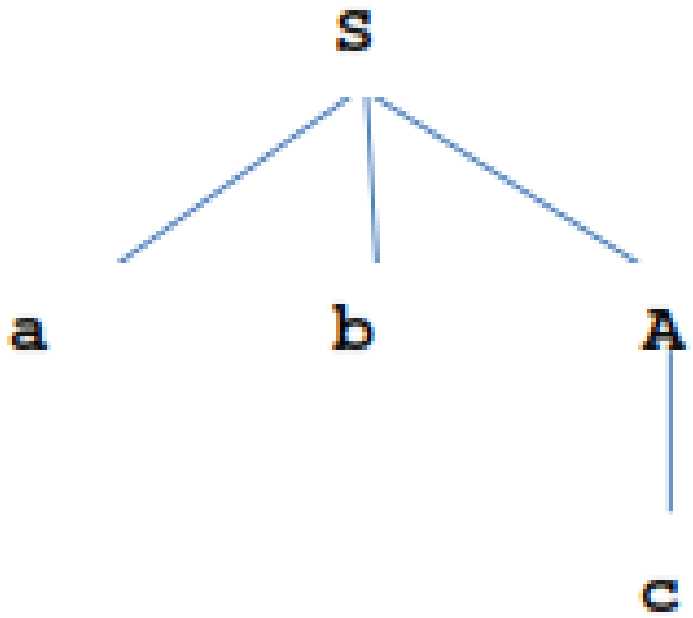



7. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 246

$S \rightarrow abA$
$A \rightarrow cd \mid c \mid e$

- ✓ Since it does not match $ab$, it backtracks and tries the alternative

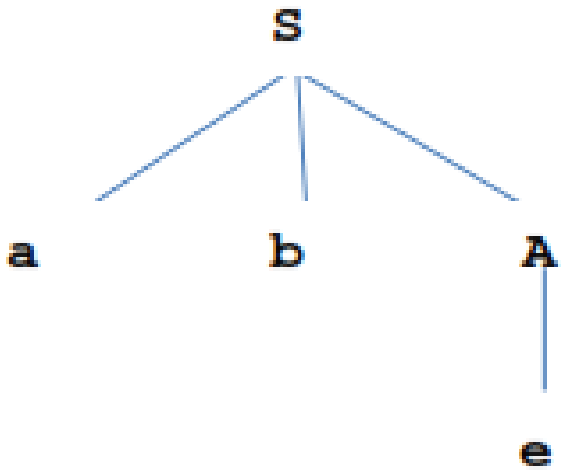
  $A \rightarrow c$ as shown below[8]:




8. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 246

$S \rightarrow abA$
$A \rightarrow cd \mid c \mid e$

- ✓However, the parse tree does not match ***ab***; the parser backtracks and tries the alternative $A \rightarrow e$ as shown below:-




- ✓With this, it finds a match and thus parsing is completely successful[9].

9. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 247

- ✓Generally, using recursive descent parsing, one can write a procedure for each non-terminal in the language definition:

i). Where the **right hand side** contains alternatives, the **next symbol to input** provides the **selector for a switch statement**, each **branch** of which represents **one alternative**[10].

10. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 89

ii). If ***e*** is an alternative, it becomes the **default** of the switch.

iii). Where a **repetition occurs**, it can be implemented **iteratively**.

iv). The sequence of actions within each branch, possibly one, consists of an inspection of the lexical token for each terminal, and a **procedure** will be **assigned to a local or global variable** as required by semantics of the language[11].

11. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 242

**Example:** The following is a recursive descent parser for the grammar shown below[12]:

- ✓ $T' \rightarrow T\$$
- ✓ $T \rightarrow R$
- ✓ $T \rightarrow aTc$
- ✓ $R \rightarrow \varepsilon$
- ✓ $R \rightarrow bR$

12. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 244

- ✓ $T' \rightarrow T\$$
- ✓ $T \rightarrow R$
- ✓ $T \rightarrow aTc$
- ✓ $R \rightarrow \varepsilon$
- ✓ $R \rightarrow bR$

```
parseT′() =
        if next = ′a′ or next = ′b′ or next = ′$′ then
                parseT() ; match(‘$′)
        else reportError ()
parseT() =
        if next = ′b′ or next = ′c′ or next = ′$′ then
                parseR()
        else if next = ′a′ then
                 match(′a′) ; parseT() ; match(′c′)
        else reportError()
parseR() =
        if next = ′c′ or next = ′$′ then
                (* do nothing *)
        else if next = ′b′ then
                match(′b′) ; parseR()
        else reportError()
```

[13]

13. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 89

# B. Predictive Parsing

- ✓ Predictive parsing is a special form of recursive descent parsing in which the **current input token unambiguously determines the production to be applied at each step** i.e. they are ***backtrack free***[14].

14. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 89

# Non-recursive Predictive – LL(k) Parsing

- ✓ ***LL*** Stands for: **Left to right scan; Left most derivation**.
- ✓ This is a parsing strategy which involves **building tables instead of writing code.**
- ✓ It is also possible to **automatically** create tables.
- ✓ These parsers scan over the input stream using a prefix of tokens so as to identify production to be applied, where ***k* is the length of the prefix**[15].

15. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 132

# Non-recursive Predictive – LL(k) Parsing…

- ✓ The language accepted by these parsers is called ***LL(k)***, where ***k* is the length of the prefix**.
- ✓ ***LL*** Stands for: **Left to right scan; Left most derivation**.
- ✓ The length of the prefix to be considered will be ***k* = $1^{16}$.**

16. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 226

# Non-recursive Predictive – LL(k) Parsing…

- ✓ The construction of an ***LL*(1)** parser for the grammar **<$V_N$ ,$V_T$, P, S>** requires computation of the following properties:-

I. **FIRST(A)** = {**u** | **u** ∈ **$V_T$** *and* ***A*** *can derive a string starting with* ***u***} [17]

17. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 227

**II. FOLLOW(A)** = $\{u \mid u \in V_T$ *and* **S** *can derive* a string like $Au\beta\}$

**III. FIRST($\alpha$)** = $\{u \mid u \in V_T$ *and* $\alpha$ *can derive a string* like $u\beta\}$

***Recall that:-***

- ✓ ***FIRST (A)*** *is the set of terminals that* ***can start*** *a string derivable from* ***A****;*
- ✓ ***FOLLOW (A)*** *is the set of terminals that* ***can follow*** *an occurrence of* ***A****,*
- ✓ ***FIRST ($\alpha$)*** *is the set of terminals that* ***can start*** *a string derivable from* $\alpha$. [18]

18. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 227

# LOOKAHEAD

- ✓ For a production rule, $\boldsymbol{A \rightarrow \alpha}$, $\boldsymbol{LOOKAHEAD(A \rightarrow \alpha)}$ is defined as the set of terminals which can **appear next in the input** when recognizing a production rule $\boldsymbol{A \rightarrow \alpha}$.
- ✓ Thus, a production rule's LOOKAHEAD set **specifies the tokens which should appear next in the input before a production is applied[19].**

19. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 87

**To build $LOOKAHEAD(A \rightarrow \alpha)$:**

i). Put $FIRST(\alpha) - \{e\}$ in $LOOKAHEAD(A \rightarrow \alpha)$

ii). If $e \in FIRST(\alpha)$

then put $FOLLOW(A)$ **in** $LOOKAHEAD(A \rightarrow \alpha)$

✓A grammar $G$ is $LL(1)$ **iff**, for each set of production, $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$

$LOOKAHEAD(A \rightarrow \alpha_1)$
$LOOKAHEAD(A \rightarrow \alpha_2)\ldots$

$LOOKAHEAD(A \rightarrow \alpha_n)$ are **all pairwise disjoint**[20]

20. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 87

**Facts about *LL*(1) grammar:**

i). No left recursive grammar is *LL*(1).

ii). No ambiguous grammar is *LL*(1).

iii). Some languages have no *LL*(1) grammar.

iv). An ***e*-free grammar** where each alternative expansion for ***A*** begins with a distinct terminal is **a simple *LL*(1) grammar**[21].

21. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 87, 88

**Example 1: Given the grammar** S →aS|a, find whether it is $LL(1)$ grammar.

**Solution:**

✓$FIRST(S) = \{a\}$

✓Therefore, $LOOKAHEAD(S \rightarrow aS) = LOOKAHEAD(S \rightarrow a) = \{a\}$

Hence it is **not** $LL(1)$grammar

**Note:** FOLLOW sets are not required since empty sets are not required in the grammar[22].

22. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 87

**Example 2: Consider the following grammar:**

$S \rightarrow aAa$

$S \rightarrow b$

$A \rightarrow ab$

$A \rightarrow e$

Verify whether the grammar is $LL(1)$ or not.

Solution:

$LOOKAHEAD(S \rightarrow aAa) = FIRST(a) = \{a\}$
$LOOKAHEAD(S \rightarrow b) = FIRST(b) = \{b\}$
$LOOKAHEAD(A \rightarrow ab) = FIRST(a) = \{a\}$

$LOOKAHEAD(A \rightarrow e) = FOLLOW(A) = \{a\}$

Therefore the grammar is **not $LL(1)$** since **$a$** is in **both $A \rightarrow ab$** and **$A \rightarrow e$**[23].

23. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 233

# LL (1) Parse Table Construction

## Method 1:

- ✓ Once the **first and follow states have been computed,** it is possible to construct $LL(1)$ parse table $M$ that **maps pairs of non-terminals and terminals to production** using the following method:

**Input:** Grammar $\boldsymbol{G}$

**Output:** Parsing table $\boldsymbol{M}$[24]

24. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 225

**Method:**

1. $\forall$ $production\ A \rightarrow \alpha$
   $\boldsymbol{\forall a \in LOOKAHEAD\ A \rightarrow \alpha}$
   $\boldsymbol{add\ A \rightarrow \alpha\ to\ M\ [A, a]}$
2. Set each unidentified entry of **M** to error
3. If $\boldsymbol{\exists M[A, a]}$ with **multiple entries** then the grammar is **not** $\boldsymbol{LL(1)}$ [25].

25. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 297

# Example: Consider the following grammar[26]:

$S \rightarrow E$

$E \rightarrow TE'$

$E' \rightarrow +E \mid -E \mid e$

$T \rightarrow FT'$

$T' \rightarrow * T \mid /T \mid e$

$F \rightarrow id \mid num$

26. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 250

# The following is the **table** of **FIRST and FOLLOW sets**[27]**:**

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +E \mid -E \mid e$
$T \rightarrow FT'$
$T' \rightarrow * T \mid /T \mid e$
$F \rightarrow id \mid num$

| Symbol | FIRST | FOLLOW |
|---|---|---|
| S | {num, id} | {$} |
| E | {num, id} | {$} |
| E' | {+, -, e} | {$} |
| T | {num, id} | {+,-,$} |
| T' | {*, /, e} | {+,-,$} |
| F | {num, id} | {+,-,*,/,$} |
| Id | {id} | |
| num | {num} | |
| * | {*} | |
| / | {/} | |
| + | {+} | |
| - | {-} | |

27. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 251

The following is the table of **LOOKAHEAD sets:** NB: The **$ sign** is a special symbol for '*end of input*' [28].

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +E \mid -E \mid e$
$T \rightarrow FT'$
$T' \rightarrow * T \mid / T \mid e$
$F \rightarrow id \mid num$

| Production | LOOKAHEAD |
|---|---|
| S→E | {num, id} |
| E→TE′ | {num, id} |
| E′→+E | {+} |
| E′→-E | {-} |
| E′→e | {$} |
| T→FT′ | {num, id} |
| T′→*T′ | {*} |
| T′→/T | {/} |
| T′→e | {+,_,$} |
| F→id | {id} |
| F→num | {num} |

28. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 253

# The following is the **parsing table**[29]:

| Symbol | Id | Num | + | - | * | / | $ |
|---|---|---|---|---|---|---|---|
| S | S→E | S→E | | | | | |
| E | E→TE′ | E→TE′ | | | | | |
| E′ | | | E′→+E | E′→-E | | | E′→e |
| T | T→FT′ | T→FT′ | | | | | |
| T′ | | | T′→e | T′→e | T′→*T | T′→/T | T′→e |
| F | F→id | F→num | | | | | |

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +E \mid -E \mid e$
$T \rightarrow FT'$
$T' \rightarrow * T \mid / T \mid e$
$F \rightarrow id \mid num$

29. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison-Wesley Pub Co, ISBN: 0201100886 (2007) page 250

**Content Covered in Week 6: Top-Down Parsing**

At the end of the lecture, we were able to:

(i) Define Recursive Descent Parsing and Predictive Parsing

(ii)Describe backtracking technique

(iii) Describe lookahead set of terminals

# Course Text Books

1. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; *Addison- Wesley Pub Co,* ISBN: 0201100886 (2007))
2. Compiler Construction: Principles and practices; Kenneth C. Louden; *Cengage Learning*; 1st edition, ISBN-10 : 0534939724 (1997)
3. Basics of Compiler Design: Torben Mogensen; *DIKU University of Copenhagen Universitetsparken 1 DK-2100 Copenhagen DENMARK* (2007)
4. Engineering a Compiler: 2nd edition; Keith D. Cooper and Linda Torczon; *Morgan Kaufmann Publishers* ISBN: 978-0-12-088478-0 (2003)
5. Compiler Design: Santanu Chattopadhyay; *PHI Learning Publishers*, ISBN 812032725X, 9788120327252 (2005)