Principles of Programming Languages(CS F301)

BITS Pilani
Hyderabad Campus

Prof.R.Gururaj
CS&IS Dept.

Lexical and Syntax Analysis (Ch.4 of T1)

BITS Pilani
Hyderabad Campus

Prof R Gururaj

Introduction

In this chapter, we get an introduction to -

- Lexical Analysis
- Parsing

The three approaches for implementing a Programming Language-

1. Compilation
2. Interpretation
3. Hybrid approach

All these implementation approaches use both lexical and syntax analyzers.

Syntax analyzers are based on a formal description of the syntax of the program.

The most commonly used descriptions are CFG / BNF.

Because:

1. BNF descriptions of PLs are clear and concise.
2. BNF descriptions can be used as direct basis for syntax analyzers.
3. Implementations based on BNF are relatively easy to maintain because of the modularity.

Analyzing Syntax has two parts-

- Lexical analysis and
- Syntax analysis.

The *Lexical Analyzer* deals with small scale language constructs such as names, numerical literals.

The *Syntax Analyzer* deals with the large scale constructs such as expressions, statements, and program units.

Reasons for separating Lexical analysis from Syntax analysis:

1. Simplicity
2. Efficiency
3. Portability

Lexical Analysis

It is essentially pattern matching.

The process of pattern matching has been a traditional part of computing.

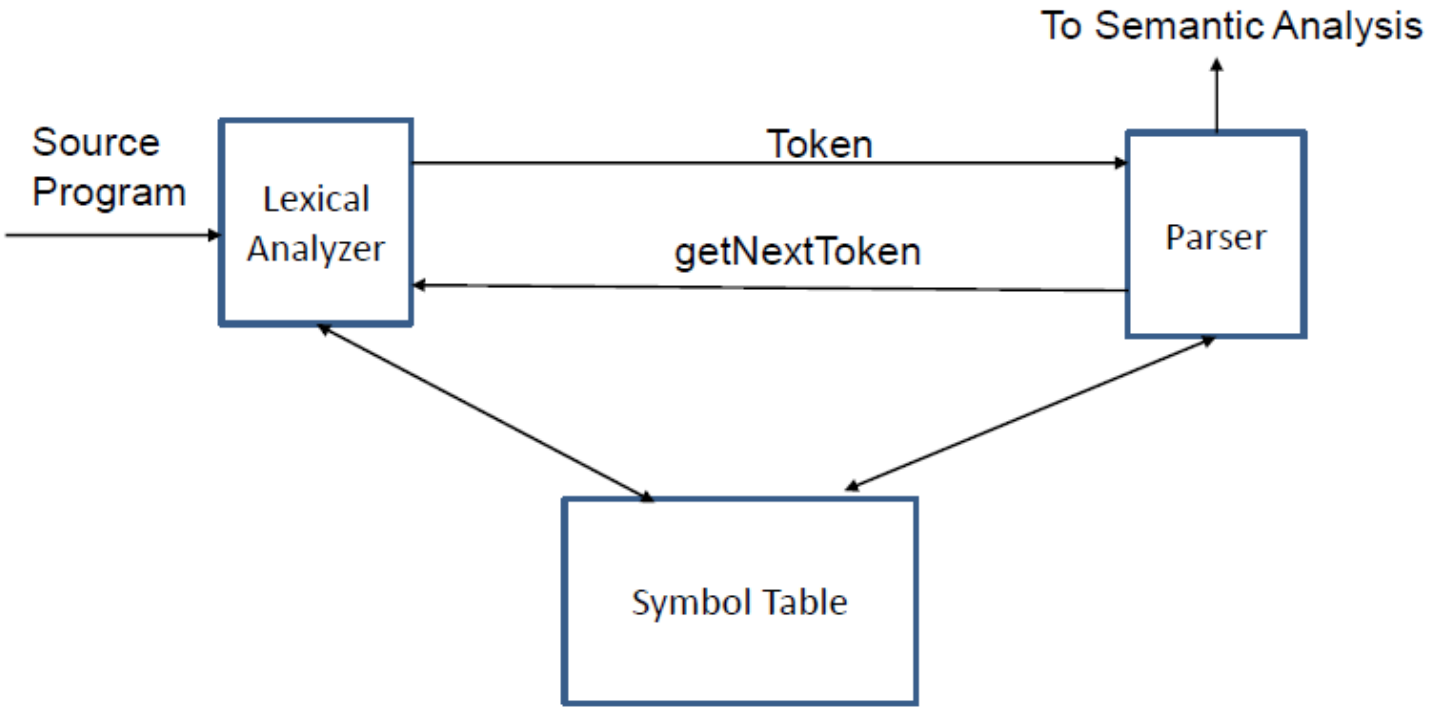
A Lexical Analyzer serves as a front-end of the Syntax Analysis.

Technically, Lexical Analyzer is part of Syntax Analyzer.

The Lexical analyzer collects characters and forms lexemes.

Each lexeme belongs to a category (token).

- The Lexical Analyzer(LA) extracts lexemes from the given input string and produces tokens.
- The Syntax analyzer (SA) module calls lexical analyzer.
- Hence each call to LA by SA returns a lexeme and its token.
- Lexical analysis process includes skipping comments and white spaces outside lexemes.
- Important: Lexical analyzer detects syntactic errors in tokens- identifiers, numeric literals etc.


To Semantic Analysis
Source Program
Lexical Analyzer
Token
getNextToken
Parser
Symbol Table

```
result = oldsum - value / 100;
```

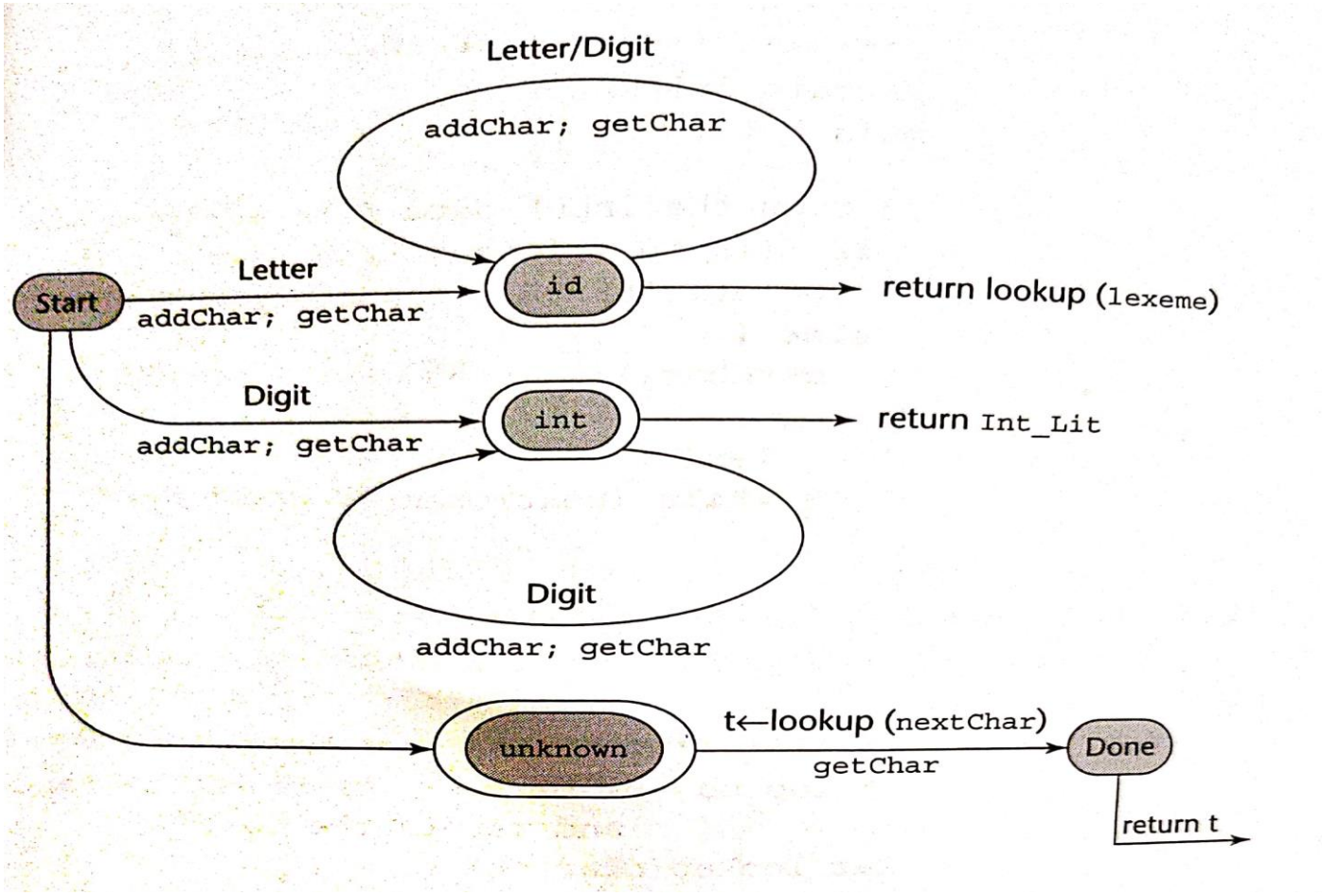
Following are the tokens and lexemes of this statemen

Token	*Lexeme*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

Approaches for building Lexical Analyzer:

1. Write a formal description of the token pattern of language using descriptive language related to Regular expression. Then tools like *Lex* can generate lexical analyzer programs automatically.
2. Design a transition diagram to describe token patterns of the language and write program to implement the FA.
3. Design a transition diagram to describe token patterns of the language and hand-construct a table-driven implementation of the state transition diagram.

State transition diagram for arithmetic expression including variables and integer literals as operands.



Subprograms used:

getChar() - gets the next character of input and puts it in global variable *nextChar*.

The lexeme is implemented as character string or array named *lexeme*.

addChar()- will add the *nextChar* in to string array lexeme.

lookup() - used to compute the token code for single character lexemes.

- Names (identifiers) and reserved words have same patterns.
- We can write state transitions separately to recognize for reserved words, but the size of the state diagram it becomes prohibitively large.
- It is much simpler and faster to have LA to recognize names and reserved words with same pattern and use lookup in table of reserved words to determine which names are reserved words.
- Lexical Analyzer is often responsible for initial construction of Symbol table to store info about user defined names.
- Attributes of names are filled by other parts of compiler later.

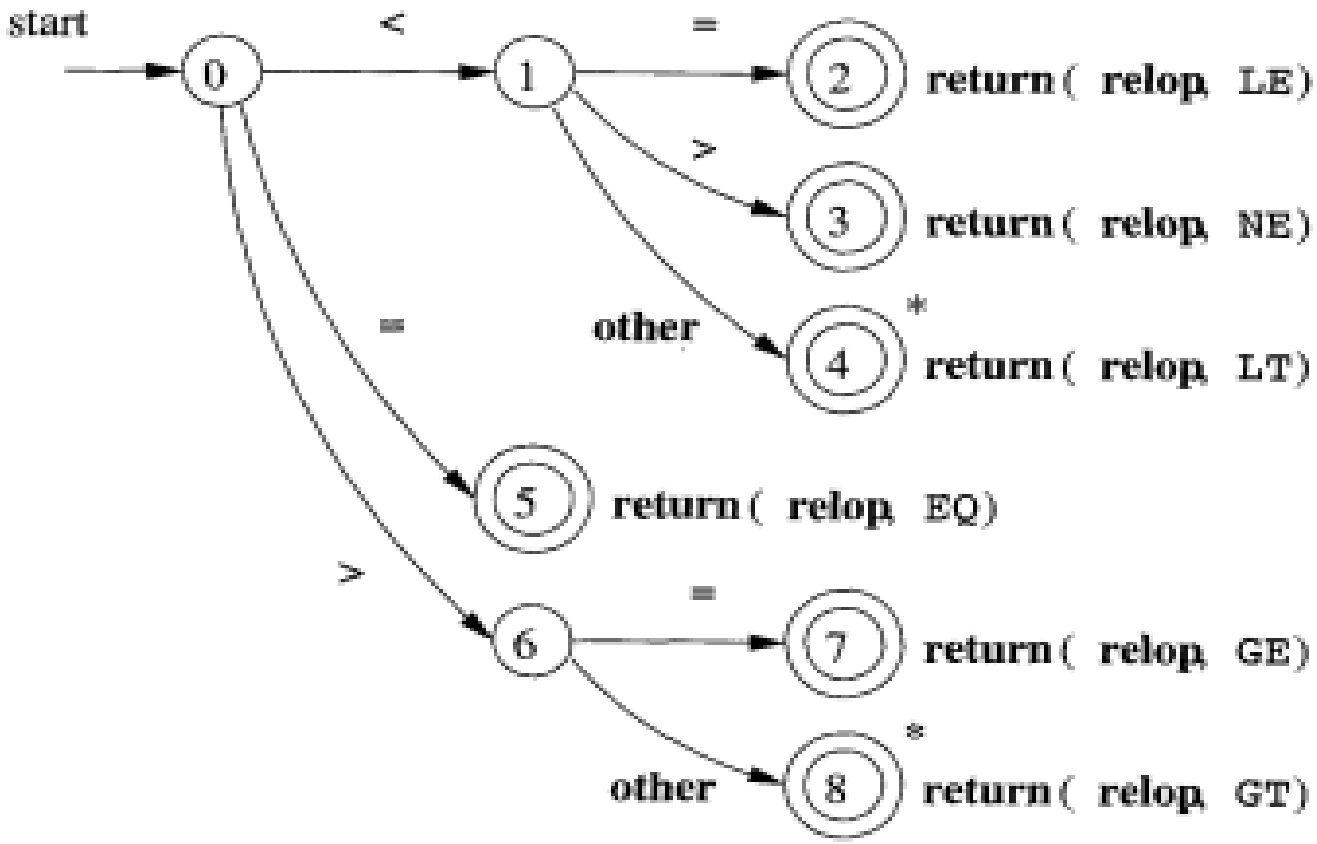
Consider the following expression:

```
(sum + 47) / total
```

Following is the output of the lexical analyzer

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

State transition diagram for Relational operators.



Parsing Problem

Syntax Analysis is also called as Parsing.

We have two approaches: Top-down and Bottom-up parsing.

Parsers construct a parse tree.

The process of derivation and parse tree include all syntactic information needed by the language.

Goals of Syntax analysis:

1.Check to syntactic correctness.

2. Produce a complete parse tree for the syntactically correct program. This parse tree is used as the basis for translation.

Notations used:

Terminals: a,b,c,…

Non-terminals : A, B, C,..

Strings of terminals: x, y, z, …

Mixed strings: α, β, δ, γ ,….

Terminal or Nonterminals: W, X, Y, Z uppercase at the end of alphabet.

Top-down Parsers

A Top-down parser builds a parse tree in preorder.

The preorder traversal of parse tree begins with root.

Each node is visited before its branches.

Branches from the particular node are followed in left-to-right order.

This corresponds to LMD.

Given a sentential form that is part of the LMD, the parser's task is to find the next sentential form in that LMD.

The general form of sentential form is-

xAα $x \epsilon \sum^*$ A is a NT and $\alpha \epsilon (\sum \cup (NT))^*$

Here A needs to be expanded using correct rule.

Determining the next sentential form is matter of choosing correct rule that has A as LHS.

This is the decision problem for Top-down parser.

Different Top-down parsers make use of different information for parsing decision.

Most parsers choose the correct rule by comparing the next input symbol with the first symbol in the RHS.

Some times two RHS involving same NT as LHS may start with same symbol then it is difficult.

A *recursive descent parser* is a coded version of a syntax analyzer based directly on the BNF descriptions of the syntax of the language.

An alternative approach is to use *parsing table* to implement BNF.

They are both called as LL parsers.

Bottom –up Parsing

Bottom-up parser constructs a parse tree by beginning at leaves and proceeding towards the root.

This corresponds to reverse of RMD.

That is the sentential forms are produced in order last to first.

The Bottom-up parsing Process:

Given a right sentential form α , the Bottom-up parser must determine which substring of the α is the RHS of rule in grammar that must be reduced to LHS to produce the previous sentential form.

For example, the first step for bottom-up parser is to determine which substring of the initial given sentence is the RHS to be replaced with corresponding LHS to get the second last sentential form in the derivation.

The process of finding the correct RHS to reduce is complicated by the fact that a given right sentential form may include more than on RHS from the grammar.

The correct RHS is called the *handle*.

Ex: 2

S→aAc

A→ aA | b

S=> aAc =>aaAc => aabc

The bottom up parser finds the handle of the given right sentential form by examining the symbols on both sides of the possible handles.

The most common Bottom-up parser algorithms are of LR family.

L- left to right

R- RMD

Usually the complexity of general parsing is $O(n^3)$

But less general parsers used in compilers can do it O(n).

Recursive descent parsing (Top-down)

As the name suggests, it consists of collection of subprograms, many of which are recursive in nature.

It produces parse tree in top-down order.

This recursion is reflection of the nature of PLs which include several different kinds of nested structures.

Ex: statements are nested.

And we see that the syntax is naturally described using recursive grammar rules.

EBNF is ideally suited for Recursive descent parsing.

A Recursive descent parser has a subprogram for each NT in its grammar.

A recursive descent subprogram for a rule with a single RHS is relatively simple.

For each terminal symbol in the RHS, the terminal is compared with next token.

If they do not match, it is syntax error.

If they match, the lexical analyzer is called for the next input token.

For each NT, the parsing subprogram for the NT is called.

Recursive descent parsing

Grammar:

$S \rightarrow aAbB$

$A \rightarrow aA \mid c$

$B \rightarrow bB \mid a$

We will have one procedure for each Non-terminal S, A, and B;
and one main procedure parse()
to start the parsing process and call S()
The procedure associated with starting NT.

main procedure parse() // parser program

```
parse( )
{  lexeml);
   S();
}
```

NOTE:

function lexeme()
returns the next input token

nextToken – is a variable that stores the token returned by lexeme() function

Recursive descent parsing

```
S()   // procedure for NT S
{ if (nextToken = 'a')
  { lexeme();
    A();
    lexeme();
    if (nextToken = 'b')
    { lexeme();
      B();
    }
  }
  else "error";
}
```

```
A()  // procedure for NT 'A'
{ if (nextToken = 'c')
  { lexeme();
    A();
  }
  else if (nextToken = 'c') { }
  else "error";
}

// procedure for NT 'B'
B()
{ if (nextToken == 'b')
  { lexeme();
    B();
  }
  else if (nextToken == 'a') { }
  else "error"
}
```

Check for "aacbba"

Exercise :

Give Recursive Descent Parser code for the following grammar:

S→ aAb
A→ bA | aB
B→ c

And track the subprogram calls for accepting *acacb*

S→ aAb
A→ bA | aB
B→ c

One valid String is:

S → aAb
→ abAb
→ abaBb
→ abacb

```
S() // procedure for 'S'
{   if (nextToken == 'a')
    {lexeme();
     A();
     lexeme();
    else if (nextToken == 'b')
        Accepted
    else "error";
}
```

```
A() // procedure for A
{ if (nextToken == 'b')
   { lex();
     A();
   }
else if (nextToken == 'a')
   {  lex();
      B();
   }
 else "error";
}
```

```
B() // procedure for 'B'
{  if (nextToken == 'c'){}
   else "error";
}
```

```
// main procedure
parse()
{ lexeme();
   S();
}
```

LL Grammar Class

Let us understand the limitation of the recursive descent approach.

1. Left Recursion.

Direct Recursion:

A--> A+B

How to eliminate.

Indirect recursion:

A--> BaA

B--> Ab

There are algorithms to remove indirect recursion . But we don't discuss now.

Left recursion

innovate achieve lead

E→E+T

Creates problems. This is known as *Left Recursion*.

Whenever we have production of the form-

$A \rightarrow A\alpha_1$, $A \rightarrow A\alpha_2$, …, $A \rightarrow A\alpha_n$, and

$A \rightarrow \beta_1$, $A \rightarrow \beta_2$, …. , $A \rightarrow \beta_m$

Replace these by

$A \rightarrow \beta_1 A'$, $A \rightarrow \beta_2 A'$, …, $A \rightarrow \beta_m A'$, and

$A' \rightarrow \alpha_1 A'$, $A' \rightarrow \alpha_2 A'$, …, $A' \rightarrow \alpha_n A'$, and

$A' \rightarrow e$

E→E+T

E→T

Can be replaced by

E→ TE'

E'→ +TE'

E'→e

Exercise :

Now eliminate Left recursion from the following Grammar.

S→ SabA | Sc

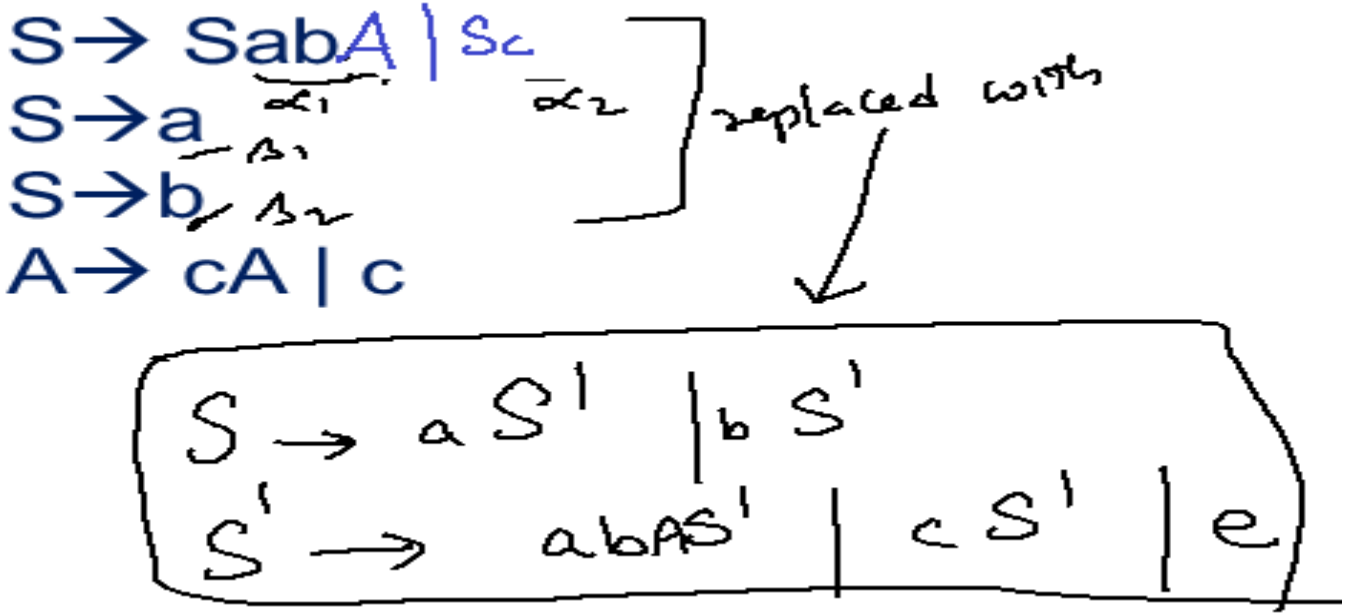
S→a

S→b

A→ cA | c

This Grammar can generate:

S→SabA→ScabA→bcabA→ *bcabc*



Now with the modified grammar, can we get “bcabc”

S→bS' → bcS' → bcabAS' → bcabA → *bcabc*

Hence both grammars generate same language.

Can a Recursive Descent Parser always choose a correct RHS based on the first symbol in RHS?

In first place it must be non-left recursive.

Test: Pairwise disjoint test.

Compute the FIRST(α) where α is the RHS.

A→ aB FIRST (aB)= {a}

A→ b FIRST (b) = {b}

B→cd |e

Here there is no problem.

A→aB

A→b

B→cd |e

FIRST(aB) is {a}

FIRST(b) is {b}

Hence first of (A)= {a,b}

Two productions of A do not start with same symbol hence no problem.

A→ abb | acb

FIRST(abb) match with one of symbols in FIRST(acb)

Hence it is a problem.

Difficult to decide which RHS to use.

We can apply *left factoring*.

A→ aM

M→ bb | cb

This may not work always.

In that case we will have to revise the grammar rules.

Bottom-up parsing

Consider the following Grammar.

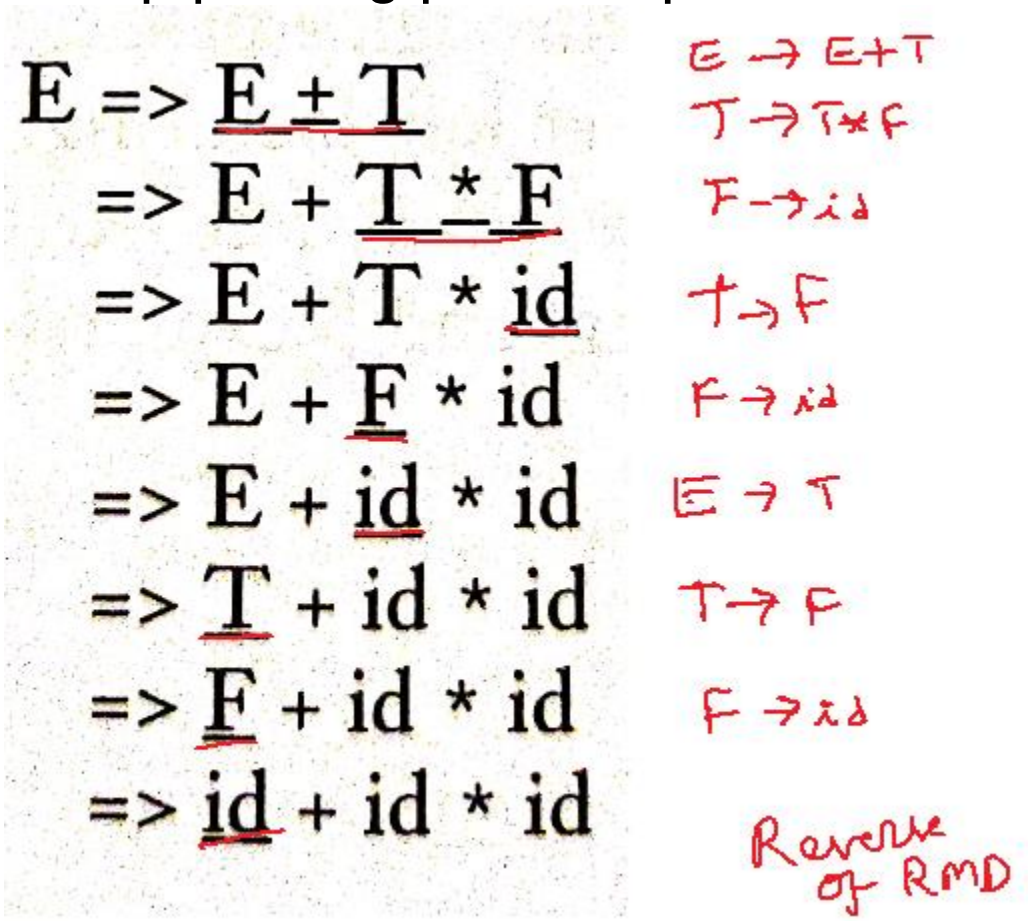
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

This is left recursive.

LL parser cannot handle this.

But bottom-up parser has no issues with Left recursion.

Bottom-up parsing process produces the reverse of RMD.



Bottom-up parsing

So the bottom-up parser starts with the last sentential form and produces the sequence of sentential forms from there until all that remains is the starting NT.

In each step, the task of the parser is to find the specific RHS (the handle), in the sentential form that must be rewritten (replaced with LHS) to get previous sentential form.

Some times, a Right sentential form may include more than one RHS in it.

Example: E+T*id { we have E→ E+T; F→ id}

But if we chose E+T as the handle it gives E*id, which is not a legal right sentential form.

The handle of a right sentential form is unique.

The task of a bottom-up parser is to find the handle of any given right sentential form.

Definition: β is the **handle** of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$

Shift-reduce Algorithm

The bottom-up parser are often called shift-reduce algorithms.

A parser is a PDA.

The input is examined from left-to-right, one symbol at a time.

LR parsers

Many bottom-up parsing algorithms are LR.

LR parsers are relatively small and a parsing table is built for specific language.

Original LR parser(canonical LR parser, 1965):

Required large computational resources to build the table.

Many variants came up in 1975 which require less computational resources and work on small class of grammars than canonical LR.

LR parsing table can be constructed for a grammar using tool like *yacc*.

LR parsers **can handle a large class of context-free grammars**.
The LR parsing method is a most general non-back tracking shift-reduce parsing method.
An LR parser can detect the syntax errors as soon as they can occur.
LR grammars can describe more languages than LL grammars

There are three advantages to LR parsers:

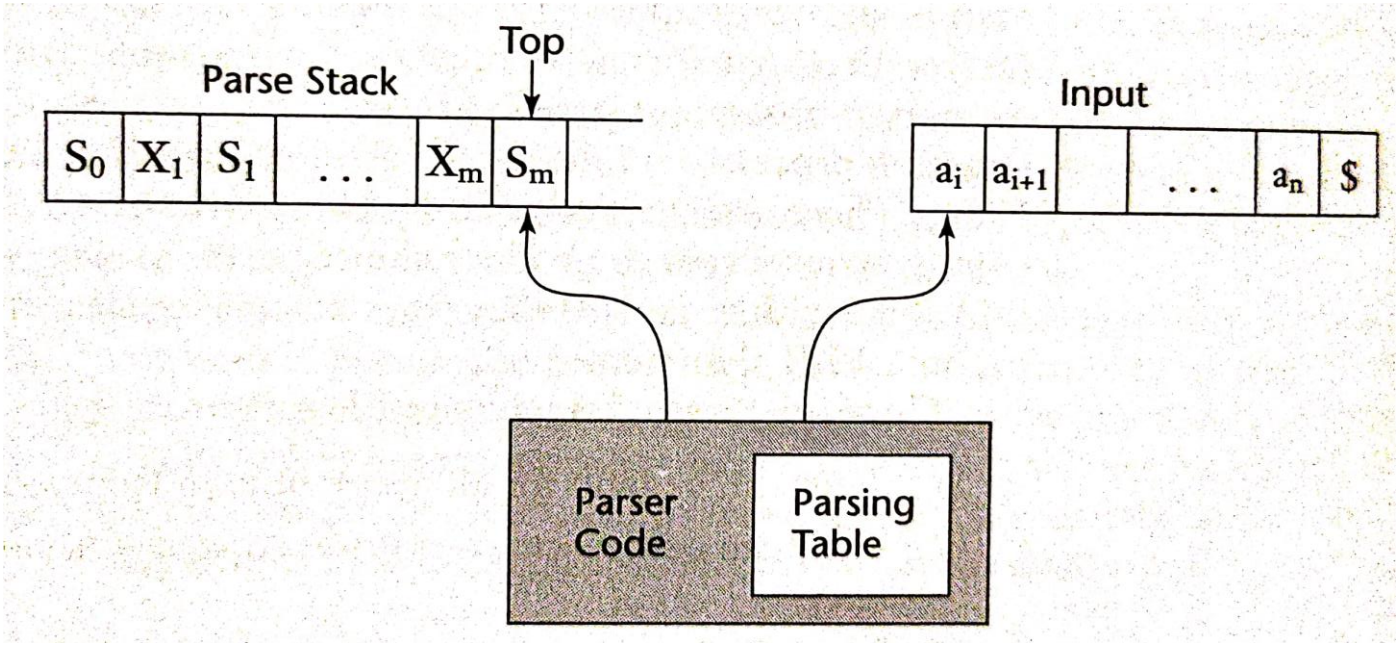
1. They can be built for all programming languages.
2. They can detect syntax errors as soon as it is possible in a left-to-right scan.
3. The LR class of grammars is a proper superset of the class parsable by LL parsers (for example, many left recursive grammars are LR, but none are LL).

The only disadvantage of LR parsing is that it is difficult to produce by hand the parsing table for a given grammar for a complete programming language.

There exist programs that generate parsing tables if input is the grammar.

configuration is a pair of strings (stack, input), with the detailed form

$$(S_0X_1S_1X_2S_2 \ldots X_mS_m, a_ia_{i+1} \ldots a_n\$)$$



Grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Parsing table

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

LR Parsing process

The parser actions are informally defined as follows:

1. The Shift process is simple: The next symbol of input is pushed onto the stack, along with the state symbol that is part of the Shift specification in the ACTION table.
2. For a Reduce action, the handle must be removed from the stack. Because for every grammar symbol on the stack there is a state symbol, the number of symbols removed from the stack is twice the number of symbols in the handle. After removing the handle and its associated state symbols, the LHS of the rule is pushed onto the stack. Finally, the GOTO table is used, with the row label being the symbol that was exposed when the handle and its state symbols were removed from the stack, and the column label being the nonterminal that is the LHS of the rule used in the reduction.
3. When the action is Accept, the parse is complete and no errors were found.
4. When the action is Error, the parser calls an error-handling routine.

Depicting stack operation of LR parser

Stack	*Input*	*Action*
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

Exercsise

Grammar

1. S→CC
2. C→aC
3. C→d

STATE	ACTION			GOTO	
	a	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Give parsing steps to accepting ***aadd*** using the grammar and parsing table

Stack	input	Action
0	aadd$	S3
0a3	add$	S3
0a3a3	dd$	S4
0a3a3d4	d$	R3 GOTO(B,C)
0a3a3C8	d$	R2 GOTO(3,C)
0a3C8	d$	R2 GOTO(0,C)
0C2	d$	S7
0C2d7	$	r3 goto[2,C]
0C2C5	$	r1 goto[0,S]
0S1	$	Accept

Grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Parsing table

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Exercise

Using the LR parsing table given, give the process of accepting the string *(id)* generated by the grammar with a table that includes input string and stack and actions.

Stack	input	Action
0	(id) $	S4
0 (4	Id) $	S5
0 (4 id 5	) $	R6
0 (4 F 3	) $	R4
0 (4 T 2	) $	R2
0 (4 E 8	) $	S11
0 (4 E 8) 11	$	R5
0 F 3	$	R4
0 T 2	$	R2
0 E 1	$	Accept

Homework

Using the LR parsing table given, give the process of accepting the string "*id * id* " generated by the grammar with a table that includes input string and stack and actions.

Summary Ch.4

- Intro to Lexical Analysis & Syntax Analysis
- Finite Automata
- Parsing problem (top-down & Bottom-up)
- Top-down parsers
- Recursive descent parser
- Left recursion
- Left factoring
- Bottom-up parsing (LR), Handle
- LR Parsing table
- LR parsing process