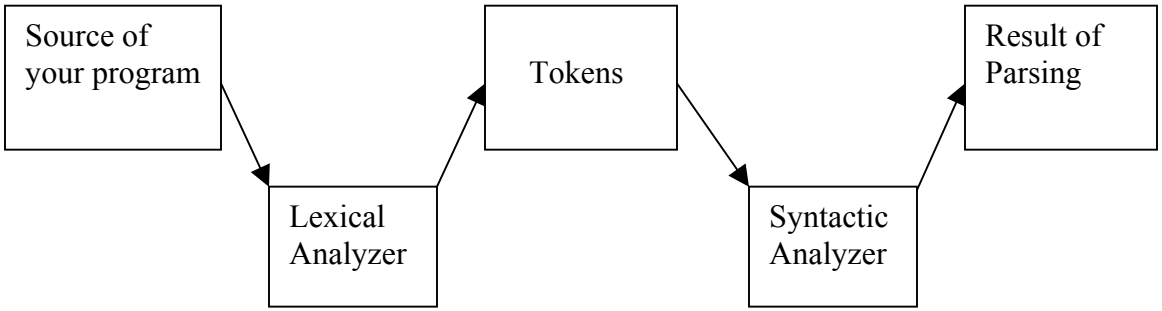
- **The Parsing Problem**
  - **Top-Down Parsing**
  - **Bottom-Up Parsing**



Parsing is the process of analyzing an input sequence in order to determine its grammatical structure with respect to a given BNF grammar.

A **parser** is a computer program that carries out this task.

Goals of the parser

Given the source of your program,

- Check the input program to determine whether it is syntactically correct.

- Produce a complete parse tree.

Parsing strategies:

(1) top-down

- Trying to find a ***leftmost derivation*** for an input string. Equivalently, construct the parse tree from the root downward to leaf.

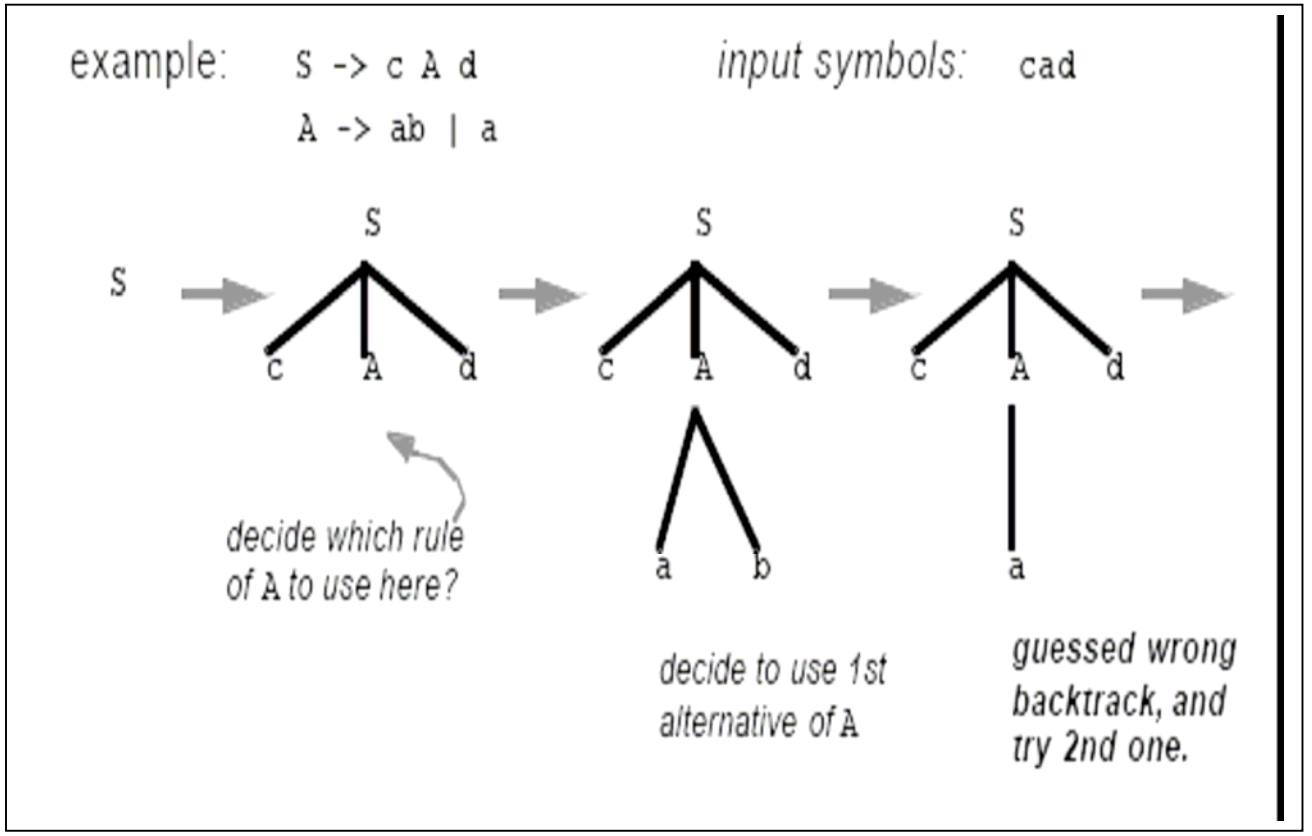
- **LL parsers** are examples of top-down parsers. It parses the input from **L**eft to right, and constructs a **L**eftmost derivation of the sentence (Hence LL).

Example:

<E>→ <F> <EP>
<EP> → + <F>
<F> → id | num.

Parse id+id

<E>→ <F> <EP>
<E>→ id <EP>
<E>→ id + <F>
<E>→ id + id



Example of top-down parsing: ***Recursive-descent parsing***

It consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down order.

• There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal.

• EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals. (minimize the number of subprograms)

<E>→ <F> <EP>
<EP> → + <F>
<F> → id | num.

Three functions: E( ), EP( ) and F ( ) corresponding to the three nonterminals <E>, <EP> and <F> in the grammar.

Recursive-descent parsing

<E>→ <F> <EP>
<EP> → + <F>
<F> → id | num.

Parse id+id

```
void main() {
   t = next_token(); // initialize with
                     // the first token
   E(); // start parse

   }

 viod E() {      // <E>→ <F> <EP>
     F();
     EP();
 }
```

```
void EP()      // <EP> → + <F>
{
    if (t == PLUS) {
     t =next-token();
     F();
}

void F() {                  // <F> -> id | num)
    if (t == ID || t == NUM) {
    t = next-token(); }
} else error("expected identifier or
number”;}
```

A grammar for simple expressions:

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | (<expr>)

• Assume we have a lexical analyzer named **lex( )**, which puts the next token code in **nextToken**.

– For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error.

– For each nonterminal symbol in the RHS, call its associated parsing subprogram.

```
/* Function expr parses strings in the language generated by the rule:
<expr> → <term> {(+ | -) <term>}
*/
void expr() {
/* Parse the first term */
  term();
  while (nextToken == PLUS_CODE || nextToken == MINUS_CODE){
    lex();
    term();
  }
}
```

The parsing subprogram for **<term>** is similar to that for <expr>

```
/* <term> -> <factor> {(*|/) <factor>}
void term() {
      factor();
      while (nextToken == TIMES_CODE||nextToken == SLASH_CODE)  {
            lex();
            factor();  }}
```

• A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse

–The correct RHS is chosen on the basis of the next token of input

–If no match is found, it is a syntax error

/* Function factor

<factor> -> **id | (<expr>)** */

```
void factor() {
    /* Determine which RHS */
     if (nextToken) == ID_CODE)
           lex();
     else if (nextToken == LEFT_PAREN_CODE) {
           lex();
           expr();
           if (nextToken == RIGHT_PAREN_CODE)
                 lex();
           else
                 error();}
    else error(); /* Neither RHS matches */
}
```

Exercise: (***Recursive-descent parsing***)

<E> -> <T>< EP>
<EP> -> + <T>< EP> | - <T>< EP>
<T> -> <F>< TP>
<TP> -> * <F> <TP> | / <F> <T>
<F> -> num | id

Here is a C program that parses this grammar:

```
E() {       // <E> -> <T>< EP>
    T();
    EP(); }

EP() {    // EP -> + <T>< EP> | -<T>< EP>
  if (current_token == PLUS)
     { next_token(); T(); EP(); }
  else if (current_token == MINUS)
     { next_token(); T(); EP(); };
}

T() {      // <T> -> <F>< TP>
   F();
   TP(); }

TP() {   // TP -> * <F>< TP> | / <F>< TP>
  if (current_token == TIMES)
     { next_token(); F(); TP(); }
else if (current_token == DIV)
     { next_token(); F(); TP(); };
}

F() {      // F -> num | id
  if (current_token == NUM || current_token == ID)
     next_token();
  else error();
}
```

- For each nonterminal we write one procedure;
- For each nonterminal in the RHS of a rule, we call the nonterminal's procedure;
- For each terminal, we compare the current token with the expected terminal.
- If there are multiple productions for a nonterminal, we use an if-then-else statement to choose which rule to apply.

The restriction of the recursive descent parsing

–The Left Recursion Problem

• If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser

A->A+B

Indirect left recursive poses the same problem as direct left recursion.

A->BaA
B->Ab

– The inability to determine the correct RHS on the basis of next token of input,

A->aB | aAb

• **Left Factoring** --- find out the common prefixes change the production

<A> -> xy | xz

to

<A> -> x<AP>
<AP> -> y|z

Replace
<variable> → identifier | identifier [<expression>]
with
<variable> → identifier <new>
<new> → ε | [<expression>]

(2) Bottom up

Build parse trees from the leaves to the root, also called **LR** parsers.

L - Left to right scan of the input

R - generates a rightmost derivation (in reverse)

E -> E + T | T

T -> T * F | F

F -> (E) | id

Rightmost derivation (in reverse)

Parse **id+id*id**

E-> E+T

➔ E+T*F

➔ E+T*id

➔ E+ F*id

➔ E+id*id

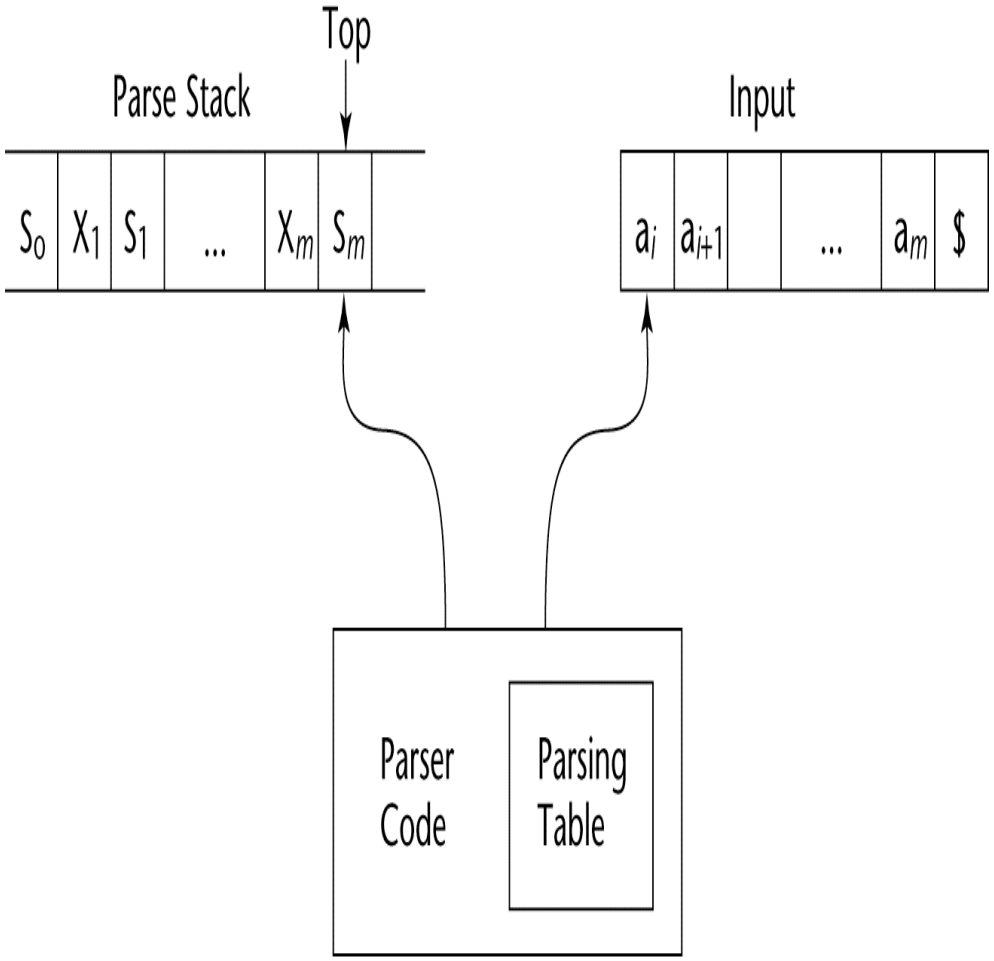
➔ T+id*id

➔ F+id*id

➔ id+id*id

The left recursive problem is acceptable to LR parsers.

Structure of an LR parser



- A LR parser uses a parse stack to keep track of the parse.

The parse stack stores the state of an LR parser ($S_0X_1S_1X_2S_2 \ldots X_mS_m$)

Ss are state symbols and the Xs are grammar symbols

- The parser is controlled by a parsing table generated from the grammar.

The parsing table has two components: an **ACTION** table and a **GOTO** table

– The ACTION table specifies the action of the parser, given the parser state and the next token

–The GOTO table specifies which state to put on top of the parse stack after a reduction action is done

To explain the parsing table workings we will use the following small grammar:

(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

The Action and Goto Table

	action						*goto*	
state	*	+	**0**	**1**	**$**		**E**	**B**
0			s1	s2			3	4
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

The action table is indexed by a state of the parser and a terminal (including a special terminal $ that indicates the end of the input stream) and contains three types of actions:

- *shift*, which is written as 's*n*' and indicates that the next state is *n*
- *reduce*, which is written as 'r*m*' and indicates that a reduction with grammar rule *m* should be performed
- *accept*, which is written as 'acc' and indicates that the parser accepts the string in the input stream.

The goto table is indexed by a state of the parser and a nonterminal and simply indicates what the next state of the parser will be if it has recognized a certain nonterminal.

The LR Parsing Algorithm

1. The stack is initialized with [**0**]. The current state will always be the state that is on top of the stack.
2. Given the current state and the current terminal on the input stream an action is looked up in the action table. There are four cases:
    - a shift s*n*:
        - the current terminal is removed from the input stream to stack, and
        - the state *n* is pushed onto the stack and becomes the current state,
    - a reduce r*m*:
        - the number *m* is written to the output stream,
        - for every symbol in the right-hand side of rule *m* a state is removed from the stack and
        - given the state that is then on top of the stack and the left-hand side of rule *m* a new state is looked up in the goto table and made the new current state by pushing it onto the stack.
    - an accept: the string is accepted
    - no action: a syntax error is reported
3. The previous step is repeated until the string is accepted or a syntax error is reported.

An Example

How a string like "1 + 1" would be parsed by LR parser.

When the parser starts it always starts with the initial state 0 and the following stack:

[**0**]

Stack	***Input***	***Action***
0	1+1$	Shift 2
0‘1’2	+ 1 $	Reduce 5 (Use GOTO[0, B])
0‘B’4	+ 1 $	Reduce 3 (Use GOTO[0, E])
0‘E’3	+ 1 $	Shift 6
0‘E’3‘+’6	1 $	Shift 2
0‘E’3‘+’6‘1’2	$	Reduce 5 (Use GOTO[6, B])
0‘E’3‘+’6‘B’8	$	Reduce 2 (Use GOTO[0, E])
0E3	$	Accept

Finally, we read a '$' from the input stream which means that according to the action table the parser accepts the input string.

Show a complete parse, including the parse stack contents, input string, and action for the string **id*(id+id)**.

1. E->E+T
2. E->T
3. T->T*F
4. T->F
5. F->(E)
6. F->id

	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			

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) $	Shift 7
0T2*7	(id + id) $	Shift 4
0T2*7(4	id+ id)$	Shift 5
0T2*7(4id5	+ id) $	Reduce 6 (Use GOTO[4, F])
0T2*7(4F3	+ id) $	Reduce 4 (Use GOTO[4, T])
0T2*7(4T2	+ id) $	Reduce 2 (Use GOTO[4, E])
0T2*7(4E8	+ id) $	Shift 6
0T2*7(4E8+6	id) $	Shift 5
0T2*7(4E8+6id5	) $	Reduce 6 (Use GOTO[6, F])
0T2*7(4E8+6F3	) $	Reduce 4 (Use GOTO[6, T])
0T2*7(4E8+6T9	) $	Reduce 1 (Use GOTO[4, E])
0T2*7(4E8	) $	Shift 11
0T2*7(4E8)11	$	Reduce 5 (Use GOTO[7, F])
0T2*7F10	$	Reduce 3 (Use GOTO[0, T])
0T2	$	Reduce 2 (Use GOTO[0, E])
0E1	$	Accept