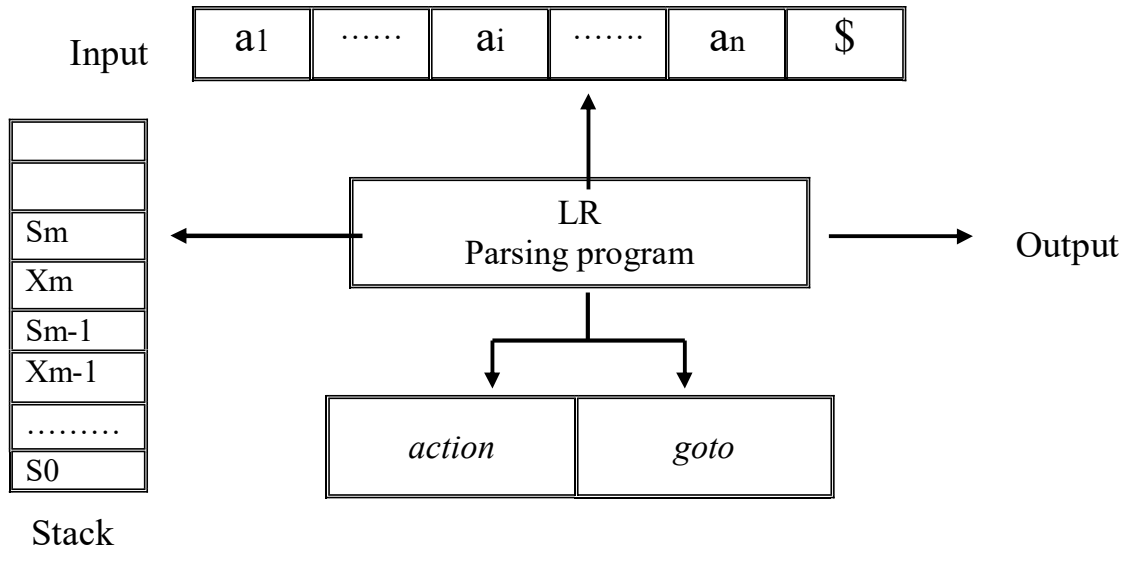
LR Parsers

This section presents an efficient Bottom-Up syntax analysis technique that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing, the "L" is for left to right scanning of input, the "R" for constructing a rightmost derivation in reverse, and "k" for the number of input symbols of lookahead that are used in making parsing decisions-when "k" is omitted , k is assumed to be 1).
LR parsing is attractive for a variety of reasons:-

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
2. The LR parsing method is the most general *nonbacktracking* shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

The schematic form of an LR parser is shown in following figure .It consists of **an Input**,**an Output**,**a Stack**,**a Driver program**,and **a Parsing table** that has two parts (*action* and *goto*) .



Model of an LR parser

There are three techniques for LR parser depending on the construct of LR parsing table for a grammar :

1. **Simple LR parser (SLR for short):**Is the easiest to implement but the least powerful of the three.It may be fail to produce a parsing table for certain grammars on which the other methods succeed.
2. **Canonical LR parser:** It is most powerful, and most expensive.
3. **Lookahead LR parser (LALR for short):**It is intermediate in power and cost between other two. The LALR method will work on most programming-language grammars and ,with some effort ,can be implemented efficiently.

The LR parsing Algorithm :- The LR program is the same for all LR parsers, only the parsing table changes from one parser to another.

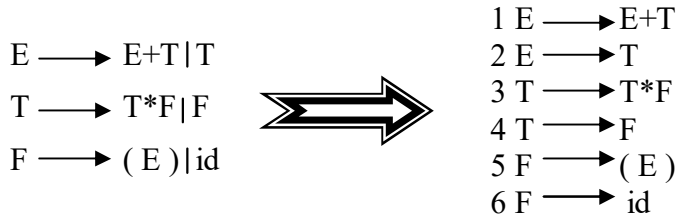
```
push the start state s0 onto the stack.
while (true) begin
    s = state on top of the stack and
    a = input symbol pointed to by input pointer ip
    if action[s,a] = shift s' then begin
       push a then s' onto the stack
       advance ip to the next input symbol
    end
    else if action[s,a] = reduce A → β then begin
       pop 2*|β| symbols off the stack, exposing state s'
       push A then goto[s',A] onto the stack
       output production A → β
    end
    else if action = accept then return
    else error()
end
```

Implementation of SLR parser:-

The SLR-parser Extremely tedious to build by hand, so need a generator. The following steps represents the main stages, which are used to build system that is used for implementing the SLR-parser:

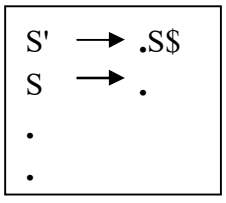
1. Input stage : In this state the grammar has been reading and the symbols of grammar (*terminals* and *nonterminals*) could be specified and each production of grammar must be on one straight line. Finally, the productions has been numbered.

For example, consider the grammar



2. Compute First & Follow stage : Through this state First & Follow could be detected for each *nonterminal*.

3. Construct DFA stage: By using a deterministic finite automaton (DFA)the SLR-parser know when to *shift* and when to *reduce.* the edges of DFA are labeled by symbols of grammar(terminals & nonterminals).In this state, where the input begins with S'(root),that means that it begins with any possible right-hand side of an S-production we indicate that by



Call this **state1** or **state0**,a productions combined with the **dot(.)** that indicates a position of parser.Firstly ,for each production in state1 we exam the symbol that occur after dot, there are three cases :

1. If the symbol is ***null*** (the dot has been occurred in the end of right side of production),then there are no new state .
2. If the symbol is **"$"** sign, then there are no new state.

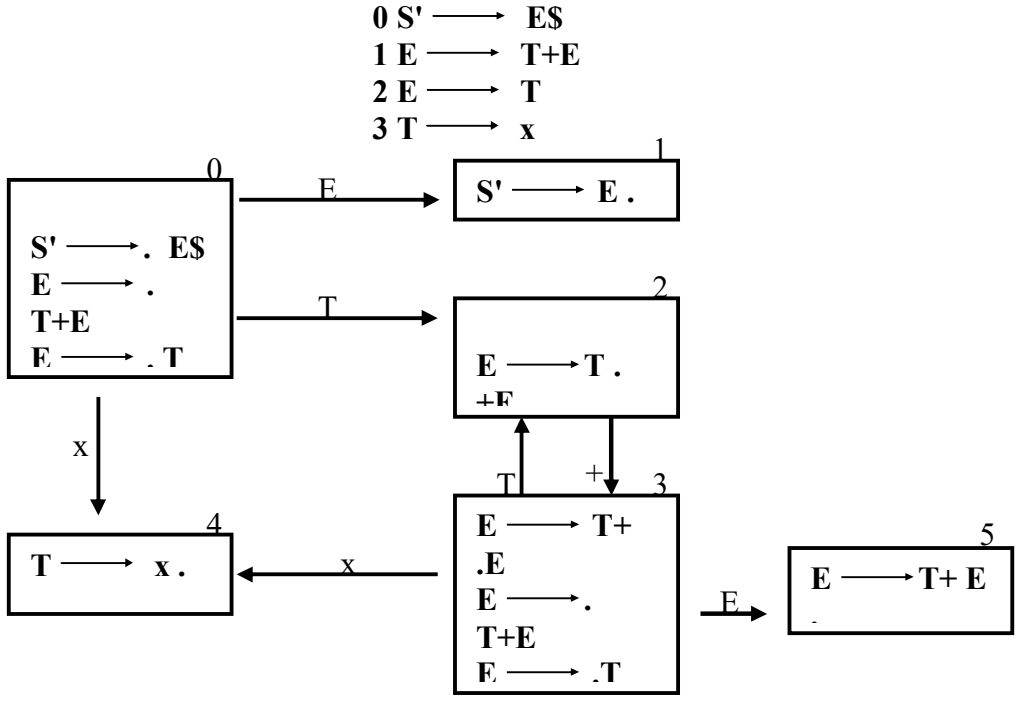
3. If the symbol is a ***terminal*** or ***nonterminal***, then there are new state, this state start with current production after the dot has been proceeded one step forward. If the symbol has been occurred after the dot(in new position)is *nonterminal* such as ***A***, then we add all possible right hand side of ***A*** to a new state, and so on.

You must know that any new state must built firstly in a buffer, and we compare it with a previous states in **DFA**, if there are no similarity situation then the new state is added to **DFA** and give it a new number equal to number of states in DFA plus one. Finally ,we repeat this steps on all new states until the **DFA** completed.

Example : consider grammar

E ⟶ T+E
E ⟶ T
T ⟶ x

Initially, it will have an empty stack, and the input will be a complete S-sentence followed by $;that is the right-hand side of the S' rule will be on the input. we indicate this as S'⟶ . S$ where the dot indicates the current position of the parser. So:



Deterministic finite automaton (DFA)

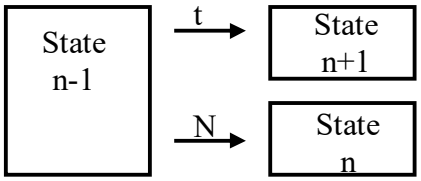
4. Construct SLR table stage: The SLR-table is a data structure consist of many rows equal to the number of the states in DFA,also many columns equal to the number of grammar symbols plus "$" sign .As know ,data structure presents fast in information treatment and information retrieve .In this stage SLR-table is constructed .this table had seen as two subtables:

1. **The Action table:** consist of many rows equal to number of states in DFA,and many columns equal to number of terminals plus "$" sign(the end of input).
2. **The Goto table:** consist of many rows equal to number of states in DFA,and many columns equal to number of nonterminals.

the elements(entries) in the SLR-table are labeled with four kinds of actions:

- Sn shift into state n
- gn goto state n
- rk reduce by production k
- a accept
- error (denoted by blank entry in the table)

For the construction of this table and the contribution the actions on the tables cells must pass to each state in DFA individually :



- Shift action & Goto action could be specified according to the edge which has been moved from the current state(n) to the new state.

  If the edge was terminal symbol (t) then

  Cell[n-1,t]= sn

  If the edge was nonterminal symbol (N) then

  Cell[n-1,N]= gn

- If there are production in current state has the form **A → ß.** (the dot in the end of right hand side, **ß** is any string),then the action is reduce

  Cell[n-1,f]=rk { **f** in Follow(A), **k** is the no. of production}
- If there are production in current state has the form **A → ß.$** {the dot occurred before $ sign, **ß** is any string },then the action is accept

  Cell[n-1,$]=a
- Finally, any empty cell in row n-1 means error action.

Repeat the above steps for each states in DFA.

state	x	+	$	E	T
0	S4			g1	g2
1			Accept		
2		S3	r2		
3	S4			g5	g2
4		r3	r3		
5			r1		

5.Implement LR Algorithm : Suppose input string is **x+x**. After insert input string the **LR-program** is executed, as follows:

Stack	Input	Action
0	x+x $	shift
0S4	+x$	Reduce by **T ⟶ x**
0T2	+x$	shift
0T2S3	x$	Shift
0T2S3S4	$	Reduce by **T ⟶ x**
0T2S3T2	$	Reduce by **E ⟶ T**
0T2S3E5	$	Reduce by **E ⟶ T+E**
0E1	$	Accept

Semantic Analysis

The semantic analysis phase of compiler connects variable definition to their uses ,and checks that each expression has a correct type.

This checking called "**static type checking**" to distinguish it from "**dynamic type checking**" during execution of target program. This phase is characterized be the maintenance of symbol tables mapping identifiers to their types and locations.

Examples of static type checking:-

1. **Type checks :** A compiler should report an error if an operator is applied to an incompatible operand.
2. **Flow of control checks:-** Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. For example, a "*break*" statement in 'C' language causes control to leave the smallest enclosing *while* , *for* , or *switch* statement ;an error occurs if such an enclosing statement does not exist.
3. **Uniqueness checks:-** There are situations in which an object must be defined exactly once. For example, in 'Pascal' language, an identifier must be declared uniquely.
4. **Name-related checks:-** Sometimes, the same name must appear two or more times. For example, in **'Ada'** language a loop or block may have a name that appear at the beginning and end of the construct. The compiler must check that the same name is used at both places.

Type system:-

The design of type checker for a language is based on information about the syntactic constructs in the language, the notation of types, and the rules for assigning types to language constructs.

The following excerpts are examples of information that a compiler writer might have to start with.

- If both operands of the arithmetic operators "*addition*", "*subtraction*", and "*multiplication*" are of type *integer* , then the result is of type *integer*.