Chapter 4: Syntax Analysis(Parsing)

The Role of a Parser: The second phase of the compilation process is syntax analysis commonly known as parsing. A parser obtains the tokens from the lexical analyzer and analyzes syntactically according to the grammar of the source language whether the string can be generated or not from the grammar i.e. the parser works with the lexical analyzer as shown in figure below.

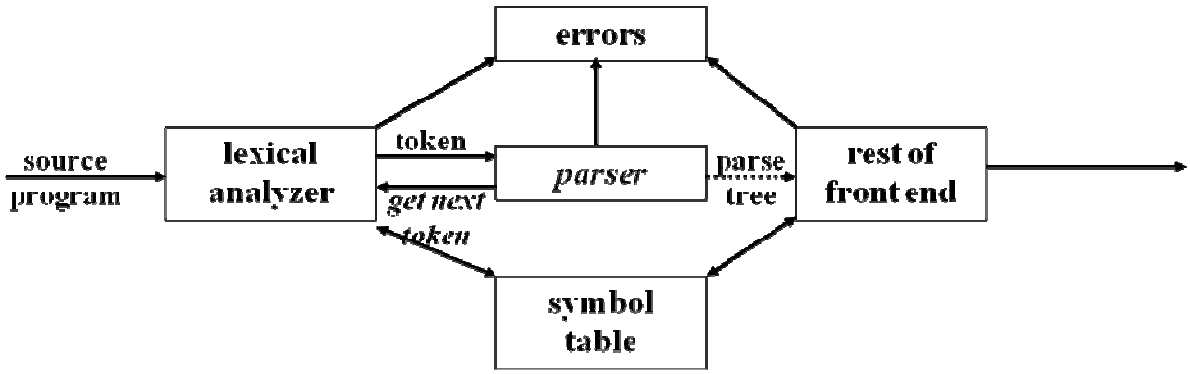


Figure: Position of parser in compiler model

A syntax analyzer(parser) is to analyze the source program based on the definition of its syntax. it works in lock-up step with the lexical analyzer(scanner) and responsible for creating a parse tree out of the source code.

- A parser implements a Context Free Grammar.
- Besides the checking of syntax the parser is responsible to report the syntax errors.
- A parser is also responsible to invoke semantic actions
  - for static semantics checking e.g. type checking of expressions, functions etc
  - for syntax directed translation of the source code to an intermediate representation
  - The possible intermediate representations outputs are
    - Abstract syntax tree
    - control-flow graphs(CFGs) with triples, three address code or register transfer list notations

Syntax Error Handling :

A good compiler should assist in identifying and locating errors. Programs may contain errors at different levels such as:

- Lexical errors: misspelling an identifier,keywords or operators. The compiler can easily recover and continue from those types of errors.
- Syntax errors: e.g. an expression with unbalanced parentheses or operator misplaced etc. Such errors are most important and can almost always recovered.
- Semantic errors(static type): important and can sometimes be recovered.

- Semantic errors(Dynamic) : Hard or impossible to detect at compile time, runtime check is needed.
- Logical errors: hard and impossible to detect by compiler. e.g. infinite loops, recursive calls etc.

Context Free Grammar:

Programming languages usually have recursive structures that can be defined by context free grammars. A CFG is defined by 4-tuples as G=(V,T,P,S) where V is set of variable symbols, T stands for set of terminal symbols, P stands for set of productions(rules) and S is a special variable called start symbol from which the derivation of each string is started.
e.g.

E → E A E | (E) | -E | id
A → + | - | * | / | ↑

Here, E and A are non terminals with E as start symbol and other symbols are terminals.

Derivation: Process of obtaining the terminal strings from the start symbol of the grammar.

- The term $\alpha \Rightarrow \beta$ denotes that β can be derived from α by applying a production of α.
- The term $\alpha \overset{*}{\Rightarrow} \beta$ denotes that β can be derived by 0 or more production rules from α
- The term $\alpha \overset{+}{\Rightarrow} \beta$ denotes that β can be derived by 1 or more production rules from αF
- If β can be derived by replacing the left most(right-most) non terminal in every derivation steps, then it is called left-most(right-most) derivation of α
  Leftmost: Replace the leftmost non-terminal symbol
  E ⇒ E A E ⇒ id A E ⇒ id * E ⇒ id * id
  Rightmost: Replace the leftmost non-terminal symbol
  E ⇒ E A E ⇒ E A id ⇒ E * id ⇒ id * id

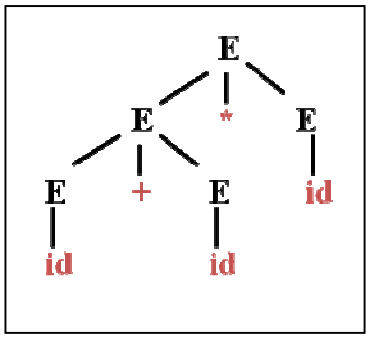
  EXAMPLE: **id * id** is a sentence of above grammar, then the derivation is
  E ⇒ E A E ⇒E * E ⇒ id * E ⇒ id * id
  E ⇒ id * id

Parse Tree: A graphical representation of the derivation of any string from the grammar

Example:

Derivation:
E ⇒ E * E
⇒ E + E * E
⇒ id + E * E
⇒ id + id * E
⇒ id + id * id

Parse Tree



Ambiguity: If a same terminal string can be derived from the grammar using two or more distinct left-most derivation(or right-most) then the grammar is said to be ambiguous. i. e. from an ambiguous grammar, we can get two or more distinct parse tree for the same terminal string.

Left –recursion: A left recursive grammar is one that has rules like $A \Rightarrow A\alpha$, for some α. Top-Down parsing can't reconcile this type of grammar, since it could consistently make choice which wouldn't allow termination and the parser moves to an infinite loop like as
$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha$ … etc. for grammar $A \rightarrow A\alpha \mid \beta$

The left recursion from the grammar can be removed as:
Left recursive grammar:

$\mathbf{A \rightarrow A\alpha \mid \beta}$

To the following:

$\mathbf{A \rightarrow \beta A'}$

$\mathbf{A' \rightarrow \alpha A' \mid \in}$

In general,
The left recursive rules like

$\mathbf{A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n}$ where no β_i begins with A. This can be converted without left recursion as

$\mathbf{A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'}$

$\mathbf{A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \in}$

Example:

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

Here E and T productions contains the left recursion so removing the recursion the grammar without the left recursion will be as

$\mathbf{E \rightarrow TE'}$
$\mathbf{E' \rightarrow + TE' \mid \in}$
$\mathbf{T \rightarrow FT'}$
$\mathbf{T' \rightarrow * FT' \mid \in}$
$\mathbf{F \rightarrow (E) \mid id}$

Parsing

Given a stream of tokens , parsing involves the process of reducing them into a non-terminal. The inpute string is said to represent the non-terminal it was reduced to.
Parsing can be either top-down or bottom up.

- Top down parsing involves generating the string starting from the first non-terminal(start-symbol) and repeatedly applying the production of the grammar.
- Bottom-up parsing involves repeatedly re-writing the input string until it ends up in the start non-terminal.

Following are the Top-down parsing algorithms

1. Recursive Descent Parsing
2. Non-recursive predictive parsing

Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

and the input string is w = cad, The recursive descent parsing algorithm can be described as below.

Consider the parsing string α = S(start Symbol)

1. Let iptr be the index of the input string and optr be the index of the output string, and initially
   iptr =optr =0
   Input: iptr(cad); output: optr(S)
2. while α[optr] is non-terminal, expand the non-terminal with its first production rule.

   Input: iptr(cad); Output: (optr)cAd
3. while w[iptr] = α[optr], increment both iptr and optr. If end of string is reached, then success.

   Input: c(iptr)ad; Output: c(optr)Ad
4. The while loop above stops if,
   a. a non terminal is encountered in α
   b. end of string is reached or
   c. if w[iptr]!= α[optr]
5. if (a) is true, then goto step 2 and expand the non-terminal with the first production.
   Input: c(iptr)ad ; Output: c(optr)abd
   Input: ca(iptr)d; Output: ca(optr)bd
6. If (b) is true then exit with success. If (c) is true then revert iptr and optr to the place they were the last expansion, and replace the non-terminal with the next production rule.
   Input: c(iptr)ad; Output: c(optr)Ad
   Input: c(iptr)ad; Output: c(optr)ad
   Input : ca(iptr)d; Output: ca(optr)d
   Input: cad(iptr) Output : cad(optr)

Exercise:
Given the following grammar
R → idS | (R) | S
S → +RS | .RS | * S | ϵ

*Then token **id** can be one of {a,b}. Determine whether the follwoing strings are in the grammar using recursive descent parsing.*

1. *a.(a+b)*.b*
2. *(b.a.b)**
3. *a.b..b.a*

Non-recursive Predictive parsing

A recursive descent parser always chooses the first available production whenever encountered by a non-terminal. This is inefficient and causes a lot of back-tracking . It is also suffers from the left-recursion problem. A predictive parser tries to predict which production produces the least chances of a backtracking and infinite looping by choosing the proper production for the derivation of the input string.

Lookahead predictive parser:

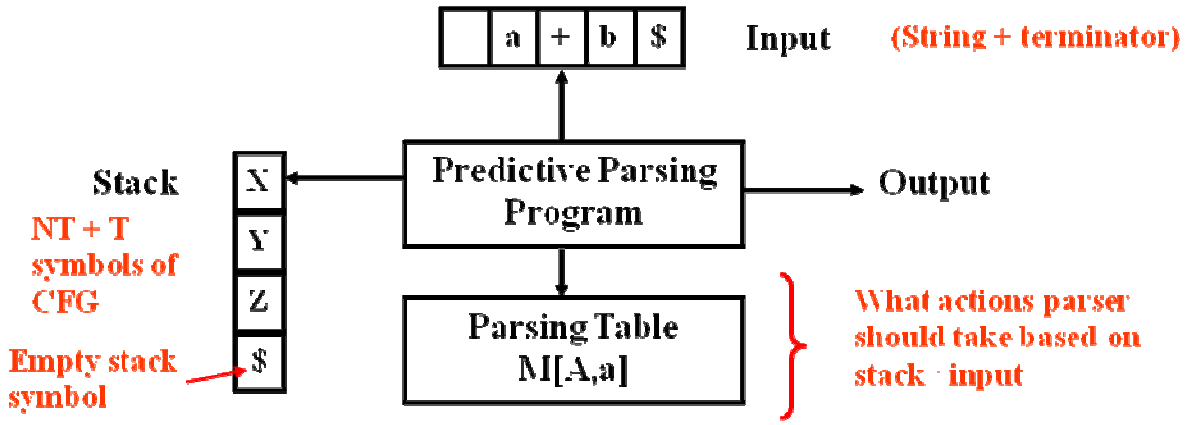
- Predictive parsing relies on information about what first symbol can be generated from a production.
- If the first symbol of a production can be a non-terminal then the non-terminal has to be expanded till we get a set of terminals.
- For any given strings of terminals and non-terminals α, FIRST(α) defines the set of all terminals that can be generated from α.

  Consider the rules:
  type → simple | id | array [simple] of type
  simple → integer | char

  then, the FIRST of each symbol can be as:
  *FIRST**(type) ={integer,char,id,array }***
  *FIRST**(simple) = { integer,char }***

  For the each terminal symbol 'a' the FIRST(a) = {a}

Non-Recursive predictive parsing



- Non-recursive predictive parsing is a table driven predictive parser comprises of an input buffer, stack, parsing table and an output stream.
- The input buffer and the stack are delimited by a special '$' symbol that denotes the end of stack or buffer.
- The parsing table is made of entries of the form M[X,a] = α, which says that if the stack points to non-terminal X and input buffer points to symbol 'a' then X has to be replaced by α, here α may be a set of terminals and non terminals or error.
- The program for the parser behaves as follows:
    1. If the stack top symbol and the input symbol is '$', the parser halts and announces successful parsing.
    2. If the stack top symbol and the input symbol matches which is not '$', the parser pops the stack and advances the input pointer to the next symbol.
    3. If the stack top symbol X is a non-terminal, then program consults entry M[X,a] of parsing table M. This entry will be either X production or an Error.
        - If it is a X production {X -> UVW } , it replaces the top of the stack X by WVU (U becomes the new top symbol)
        - If M[X,a] = Error, the parser calls an error recovery routine.

So the algorithm for the parser can be explained as below.
Input: A string w and a parsing table M for the grammar G.
Method: Initially the parser is in configuration in which $S on the stack with S on the top and w$ in input buffer.

1. Set ip to the first symbol of the input string.
2. Set the stack to $S where S is the start symbol of the grammar G.
3. Let X be the top stack symbol and 'a' be the symbol pointed by ip then
   Repeat
    a. If X is a terminal or '$' then
        i. If X = a then pop X from the stack and advance ip
        ii. else error()
    b. Else
        i. If M[X,a] = $Y_1Y_2Y_3$..... Y_k then

1. Pop X from Stack
2. Push Y_k , $Y_{k-1, \ldots..}Y_2$, Y_1 on the stack with Y_1 on top.
3. Output the production $X \rightarrow Y_1Y_2Y_3$..... Y_k

Until X = '$'.

Example:

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

Given the parsing table for the grammar G as:

Non Terminals	Inputs					
	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)		

For the input string ***id+id *id***, the moves made by predictive parser will be as follows

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E`T	id + id * id$	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`→∈
$E`T+	+ id * id$	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`→∈
$	$	E`→∈

Expend Input

The leftmost derivation for the example is as follows:

$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow id\ T'E' \Rightarrow id\ E' \Rightarrow id + TE' \Rightarrow id + FT'E'$
$\Rightarrow id + id\ T'E' \Rightarrow id + id * FT'E' \Rightarrow id + id * id\ T'E'$
$\Rightarrow id + id * id\ E' \Rightarrow id + id * id$

Parsing Table:
The parsing table M comprises the entries of the form M[X,a] =UVW meaning that if the top of the stack holds X and the input symbol 'a' is read, then X should be replaced by UVW.

- The construction of the parsing table is aided by two functions : FIRST and FOLLOW.
- If α is any string of grammar symbols, then FIRST(α) is the set of all terminals that begin the strings that can be derived from α. If $\alpha \stackrel{*}{\Rightarrow} \in$, **then ∈ is First(α).**
- For any non terminal A, FOLLOW(A) is the set of terminals 'a' that can appear immediately to the right of A in some sentential form. That is, there exist some rule of the form $S \stackrel{*}{\Rightarrow}$ **αAaβ,** for some α and β.
- If A is right most symbol in some rule, then $ is also in FOLLOW(A)

Computation of FIRST

1. For all terminals 'a' , First(a) = {a}
2. For any non terminal X, if X→ ∈ is a production rule, add ∈ to First(X) i.e FIRST(X) =FIRST(X)∪ {∈ }
3. If X is a non-terminal, for every rule of the form X→ $Y_1Y_2\ldots Y_k$, update First(X) by the following rules:

    a. FIRST(X) =FIRST(X) ∪ FIRST(Y_1).
    b. For all 1<=i<k, FIRST(X) =FIRST(X) ∪ FIRST(Y_i) if ∈ is in FIRST(Y_j) where 1<=j<i. If ∈ is in FIRST(Y_1) ∩FIRST(Y_2) ∩ ……∩ FIRST(Y_k) then FIRST(X) = FIRST(X) ∪ {∈ }

Computation of FOLLOW

Apply the following rules until nothing can be added to any FOLLOW set.
1. Place $ in FOLLOW(S), where S is the start symbol and $ is the right end marker .
2. If there is a production of the form **A→ αBβ,** then every thing in FIRST(**β**) except ∈ is placed in FOLLOW(B).
3. If FIRST(**β**) in above case contains ∈ or if there is a rule of the form **A→ αB,** then everything in FOLLOW(A) is in FOLLOW(B).

Example:
Consider the following grammar

R→**id**S |(R)S

$S \rightarrow +RS \mid .RS \mid *S \mid \in$

Here,
FIRST(R) ={ **id**, (}
FIRST(S) ={+, . , * , ∈ }

FOLLOW(R) = {) , + , . , *, $ }
FOLLOW(S) = {) , + , . , *, $ } ***from R→idS using the third rule above***

Another example:

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

FIRST(E) =FIRST(T) =FIRST(F) = {(, id }
FIRST(E') = { +, ∈ }
FIRST(T') = { * , ∈ }

FOLLOW(E) =), $ } *since E is start symbol $ is follow of E and) comes in FOLLOW(E) from the production* **F → (E)**
FOLLOW(E') = FOLLOW(E) ={), $ } *from E → TE'*
**FOLLOW(T) ={) ,+,$}** *where),$ comes from E → TE' since ∈ is in FIRST(E') + comes from FIRST(E')*
FOLLOW(T') = FOLLOW(T) = ={) ,+,$} ***since T → FT'***
FOLLOW(F) = { *,+,),$} *where * comes from FIRST(T') in T' → FT' , others come from FOLLOW(T) in T→FT'*

Example
S → i E t SS' | a
S' → eS | ∈
E → b

The FIRST and FOLLOW:

FIRST:
First(S) = { i, a }
First(S') = { e, ∈ }
First(E) = { b }

Follow(S) – Contains $, since S is start symbol
- **Since *S → i E t SS'* , put in First(S') but not ∈**
- **Since S' ⇒ ∈, Put in Follow(S)**
- **Since S' → eS, put in Follow(S') So….**
  Follow(S) = { e, $ }
- **Follow(S') = Follow(S)**
- **Follow(E) = { t }**

Construction of PARSING TABLE : Algorithm

For each production $A \rightarrow \alpha$ of the grammar do ,

1. For each terminal 'a' in FIRST(α), add $A \rightarrow \alpha$ to M[A,a].
2. (a) If $\in$ is in FIRST(α) , add $A \rightarrow \alpha$ to M[A,b] for every terminal 'b' in FOLLOW(A).
   (b)If $\in$ is in FIRST(α) and $ is in FOLLOW(A) then add $A \rightarrow \alpha$ to M[A,$]).
3. Make every undefined entry of M to be ERROR.

Example of parsing table for the grammar:

R→**id**S |(R)S
S → +RS | .RS | *S |∈
The FIRST and FOLLOW as computed above are:

FIRST(R) ={ **id**, (}
FIRST(S) ={+, . , * , ∈ }

FOLLOW(R) = {) , + , . , *, $ }
FOLLOW(S) = {) , + , . , *, $ }

The parsing table is as below:

Non Terminal	Inputs						
	id	(	)	+	.	*	$
R	R→**id**S	R→(R)S					
S			S → ∈	S → +RS S → ∈	S → .RS S → ∈	S → *S S → ∈	S → ∈

In the above example,

- FIRST(R) contains id and (so at the column with these inputs, corresponding productions of R has been entered.
- Also the FIRST(S) contains {+, . , * , ∈ } So for the symbols +, . and * the corresponding production with the first being that symbol are entered in table.
- The FIRST(S) contains ∈ so for every symbols in FOLLOW(S) the corresponding productions of S are entered .

In the above parsing table, some entries for M[X,a] is multiply-defined. If the grammar is ambiguous than the parsing table may contain the multiple entries for some M[X,a]. So construction the parsing table for the grammar, we can decide whether the grammar is ambiguous or not.

Constructing Parsing Table another Example: Grammar, FIRST and FOLLOW

S → i E t SS' | a	First(S) = { i, a }	Follow(S) = { e, $ }
S' → eS | ∈	First(S') = { e, ∈ }	Follow(S') = { e, $ }
E → b	First(E) = { b }	Follow(E) = { t }

Construction of Parsing Table:

S → i E t SS'
First(i E t SS')={i}

S → a
First(a) = {a}

E → b
First(b) = {b}

S' → eS
First(eS) = {e}

S → ∈
First(∈) = {∈}

Follow(S') = { e, $ }

Non-terminal	INPUT SYMBOL					
	a	b	e	i	t	$
S	S →a			S →iEtSS'		
S'			S' →∈ S' →eS			S' →∈
E		E →b				

Constructing Parsing Table – Another Example 3

Grammar:	FIRST	FOLLOW
E → TE' E' → + TE' | ∈ T → FT' T' → * FT' | ∈ F → (E) | id	First(E,F,T) = { (, id } First(E') = { +, ∈ } First(T') = { *, ∈ }	Follow(E) = {), $} Follow(E') = {), $} Follow(F) = { *, +,), $ } Follow(T) = { +,) , $} Follow(T') = { +,) , $}

Expression Example: E → TE' : First(TE') = First(T) = { (, id }

M[E, ( ] : E → TE'

M[E, id] : E → TE' By Rule 1

E' → +TE' : First(+TE') = +

so **M[E', +] : E' → +TE'** By Rule 1

T' → * FT' : First(*FT') = *

so, M[T',*] : T' → * FT' By Rule 1
T → FT' : First(FT') =First(F) = {(, id}
so M[T,(] : T → FT'
M[T,id] : T → FT' By rule 1
F→ (E) | id : First(F) = {(, id} so by rule 1,
M[F,(] : F → (E)
M[F,id] : F → id

(by rule 2) E' → ∈ : ∈ in First(∈) T' → ∈ : ∈ in First(∈)
M[E',)] : E' → ∈ (2.a) M[T', +] : T' → ∈ (2.a)
M[E', $] : E' → ∈ (2.a) M[T',)] : T' → ∈ (2.b)
M[T', $] : T' → ∈ (2.b)

Non Terminals	Inputs					
	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)		

Exercise: **Construct the Parsing table for the grammar below using FIRST and FOLLOW.**
S → cAd
A → ab | a

LL(1) Grammar:
A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar. The first L in LL(1) corresponds to reading the input left to right and second 'L' corresponds to the left-most derivation. The 1 in the parenthesis corresponds to a maximum lookahead of 1 symbol.

- ✓ No ambiguous or left-recursive grammar is LL(1).
- ✓ There are no general rules to convert a non LL(1) grammar into a LL(1) grammar.
- ✓ The properties of LL(1) grammar are as below:

In any LL(1) grammar, if there exists a rule of the form $A \rightarrow \alpha \mid \beta$ where α and β, are distinct then,

1. For any terminal 'a', if a is in FIRST(α) then a is not in FIRST(β)
2. Either $\alpha \overset{*}{\Rightarrow} \in$ or $\beta \overset{*}{\Rightarrow} \in$, but not both.
3. If $\beta \overset{*}{\Rightarrow} \in$, then α does not derive any string beginning with the terminal in FOLLOW(A).

Exercise:
1. Show that the following grammar is ambiguous by constructing parsing table

S → aSbS | bSaS | ∈

Bottom Up Parsing:

- ✓ Bottom up parsing attempts to construct a parse tree for an input string beginning at the leaves(the bottom) and working up towards root(top).
- ✓ The process of replacing a substring by a non-terminal in bottom up parsing is called reduction.
- ✓ The left-most reduction method corresponds to the right-most derivation of top-down parsing and the right-most reduction corresponds to the left-most derivation of top-down parsing.

Consider an example grammar
S→aABe
A→Abc | b
B → d

Now for string **abbcde** can be reduced to S by following steps
abbcde
aAbcde (replace b by A using A→ b)
aAde (replace Abc by A using A→Abc)
aABe (replace d by B using B → d)
S (replace aABe by S using S→aABe)

A substring that can be replaced by a non-terminal when it matches its right sentential form is called a **Handle.** So principle task of the bottom up parsing is to identify the handle that can be replaced by a non-terminal. Identifying the handle refers to the implementing the DFA that recognize the handle string.

Shift-Reduce Parsing

A simple bottom up parsing technique using a stack based implementation is shift-reduce parsing
A convenient way to implement a shift-reduce parser uses a stack to hold the grammar symbols and an input buffer to hold the input string **w**.
The method is described as:

1. Initially stack contains only the sentinel $, and the input buffer contains the input string ***w$.***
2. While stack not equal to $S do
    a. While there is no handle at the top of stack, shift the input buffer and push the symbol onto stack.
    b. If there is a handle on top of stack, then pop the handle and reduce the handle with its non terminal and push it on to stack.

Example:

For the above example string abbcde$, the shift reduce actions may be,

Stack	**Input**	**Action**
$	abbcde$	

$a	bbcde$	shift a
$ab	bcde$	shift b
$aA	bcde$	**reduce A→b**
$aAb	cde$	shift b
$aAbc	de$	shift c
$aA	de$	**reduce by A→Abc**
$aAd	e$	shift d
$aAB	**e$**	**reduce by B→d**
$aABe	$	shift e
$S	**$**	**reduce by** S→**aABe**

Conflict during Shift-Reduce Parsing

All grammars can not use the shift-reduce parsing since there may be conflicts during the parsing actions. For some grammars parser can not decide whether to shift or to reduce or can not decides which of several reduction to make. so the parsing action results in conflicts.

There are two kinds of conflicts in this method.

1. **Shift-reduce conflict:** The parser is not able to decide whether to shift or to reduce e.g. if A→ab | abcd are the productions of A and the stack contains $ab and the input buffer contains cd$, the parser cannot decide whether to reduce $ab by non-terminal A or to shift two more symbols before reducing.

2. **Reduce-Reduce Conflict:** In this case the parser cannot decide which sentential form to use for reduction e. g. if A →bc and B→abc are the productions of grammar and the stack contains $abc, the parser cannot decide whether to reduce it to $aA or to reduce $B.

The grammar that lead to the shift reduce parser into conflict are known as non-LR grammars
Shift reduce parser can be built successfully for LR grammars and operator grammars.
The operator grammar has the property that no production right side is ∈ or has two adjacent terminals.

e.g. E **→EAE | (E) | -E | id is not operator grammar.**
E → E+E | E-E | E*E | (E) | -E | id is an operator grammar

LR Grammars

- ✓ The class of grammars called LR(k) grammars have the most efficient bottom up parser and can be implemented for almost any programming language.
- ✓ The first L stands for left to right scan of input buffer, the second R for a right most derivation(left-most reduction) and the k stands for the maximum number of input symbols of lookahead used for making parsing decision.

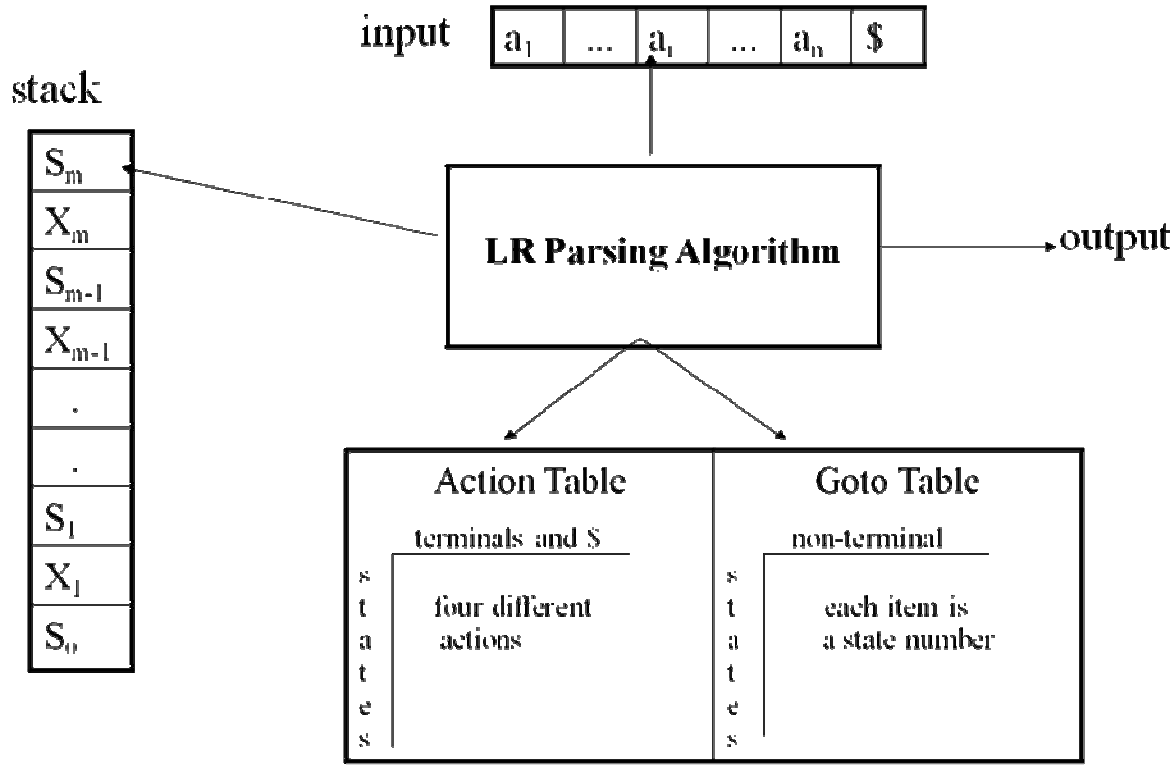
- ✓ If k is omitted, k is assumed to be 1.
- ✓ The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parser.

- ✓ It is difficult to write or trace LR parser by hand. Usually generators like Yacc(bison) is required to write LR parser.
- ✓ The LR parsing method is most general non-backtracking shift-reduce parsing method known.
- ✓ LL(1) grammar $\subset$ LR(1) grammar.

LR-Parsers

- covers wide range of grammars.
- SLR – simple LR parser
- LR – most general LR parser
- LALR – intermediate LR parser (look-head LR parser)
- SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

Structure of LR Parser:



An LR parser comprises of a stack, an input buffer and a parsing table that has two parts: action and goto. Its stack comprises of entries of the form $\mathbf{s_o\ X_1\ s_1\ X_2 s_2 \ldots X_m\ s_m}$ where every s_i is called a state and every X_i is a grammar symbol(terminal or non-terminal)

If top of the stack is S_m and input symbol is a, the LR parser consults action[s_m,a] which can be one of four actions.

1. Shift s , where s is a state.

2. Reduce using production $A \rightarrow \beta$
3. Accept
4. Error

The function goto takes a state and grammar symbol as arguments and produces a state. The goto function forms the transition table for a DFA that recognizes the viable prefixes of the grammar.

Note: Viable prefixes are the set of prefixes of right sentential forms that can appear on the stack of the parser. i. e. it is a prefix of right sentential form that doesnot continue past the end of right most handle of that sentential form.

The configuration of the LR parser is a tuple comprising of the stack contents and the contents of the unconsumed input buffer. It is of the form
$(\mathbf{s_o\ X_1\ s_1\ X_2 s_2 \ldots X_m\ s_m}\ ,\ a_i\ a_{i+1} \ldots a_n\ \$)$

1. If action$[s_m,a_i]$ = shift s, the parser executes a shift move entering the configuration $(\mathbf{s_o\ X_1\ s_1\ X_2 s_2 \ldots X_m\ s_m a_i s}\ ,\ a_{i+1} \ldots a_n\ \$)$ shifting both a_i and the state s onto stack.
2. If action$[s_m,a_i]$ = reduce $A \rightarrow \beta$, the parser executes a reduce move entering the configuration $(\mathbf{s_o\ X_1\ s_1\ X_2 s_2 \ldots X_{m-r}\ s_{m-r} As}\ ,\ a_i\ a_{i+1} \ldots a_n\ \$)$. Here s = goto$[s_{m-r},A]$ and r is the length of handle β.

   Note: if r is the length of β then the parser pops 2r symbols from the stack and push the state as on goto$[s_{m-r},A]$. After reduction , the parser outputs the production used on reduce operation. $A \rightarrow \beta$

3. If action$[s_m,a_i]$ = Accept, then parser accept i.e. parsing successfully complete and stop.
4. If action$[s_m,a_i]$ = error then call the error recovery routine.

Construction of the Parsing Table for the SLR parser (Simple LR)

- SLR parser are the simplest class of LR parser. Constructing a parsing table for action and goto involves building a state machine that can identify the handle.
- For building a state machine, we need to define the three terms: item, closure and goto

Item: *An "**item**" is a production rule that contains a dot(.) somewhere in the right side of the production. For example,* $A \rightarrow .\alpha A\beta$, $A \rightarrow \alpha.A\beta$, $A \rightarrow \alpha A.\beta$, $A \rightarrow \alpha A\beta.$ *are **items** if there is a production* $A \rightarrow \alpha A\beta$ *in the grammar.*

An item encapsulates what we have read until now and what we expect to read further from the input buffer.

The Closure operation:
If I is a set of items for a grammar G, then the closure(I) is the set of items constructed from I using the following rules:

1. Initially, every item in I is added to closure(I).
2. If $A \to \alpha.B\beta$ *is* in closure(I) and B $\to \gamma$ is a production, then add the item B $\to .\gamma$ to I if it is not already there.
   Apply this rule until no more new items can be added to closure(I).

Example:

E' →*E*
E→E+T | T
T→T*F | F
F→(E) | id

If I = {E' **→*.E } then*** *closure(I) contains the following items:*

{
E' ***→.E,***
E→.E+T ,
E→.T,
T →.T*F,
T→.F,
F→.(E),
F→.id
}

Note:

- *E'* → ***.E and all items whose dots are not at the left end(begining of RSH) are called kernel items.***
- ***All items with dot at the left end are non-kernel items except E'→ .E which is always known as kernel items.***

<u>The goto operation:</u>

In any item I, for all production of the form **A→ α.Xβ** that are in I,
Goto[I,X] is defined as the closure of all productions of the form **A→ αB.β**
In the example above, if I_o = closure({ E'→.E} then
goto[I_o,E] = I_1 = closure({E' → E. , E→E.+T }) since the closure({ E'→.E} is
{ E' →.E, E→.E+T , E→.T, T → .T*F, T→ .F, F→.(E), F→.id }

Similarly, goto[I_0,T] = closure({ E → T., T → T.*F })
goto(I_0,F) =closure({T → F. }) and so on.

The complete goto operation defines the DFA that identifies the handle.

Computing the Canonical LR(0) Collection

To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

Algorithm:

C = { closure({S'→.S}) }
repeat the followings until no more set of LR(0) items can be added to ***C***.
for each I in ***C*** and each grammar symbol X
if goto(I,X) is not empty and not in ***C***
add goto(I,X) to ***C***

The goto function is a DFA on the sets in C

Example: Compute the Canonical LR(0) items collection for the following grammar.
E→E+T | T
T→T*F | F
F→ (E) | id

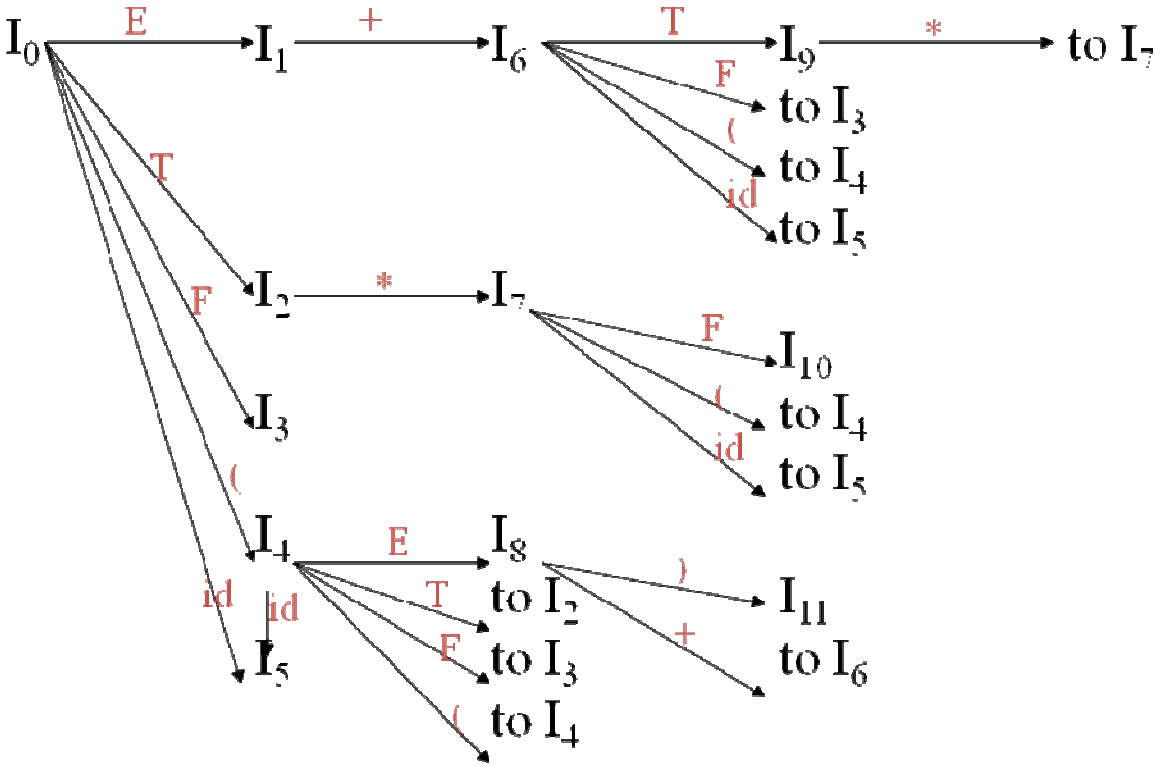
Solution: The augmented grammar is ,
E'→E
E→E+T | T
T→T*F | F
F→ (E) | id

I_0 = closure({***E'→.E***}) = {
E' →.E,
E→.E+T ,
E→.T,
T →.T*F,
T→.F,
F→.(E),
F→.id
}

goto[I_0,E] =closure({E' →E., E→E.+T}) ={ E' →E., E→E.+T} = I_1
goto[I_0,T] =closure({E→T.,T → T.*F}) = {E→T.,T → T.*F} = I_2
goto[I_0,F] =closure({T→ F.}) ={T→ F.} = I_3
goto[I_0,(] =closure({F→(.E)})
={ F →(.E), E →.E+T , E →.T , T →.T*F , T →.F, F →.(E) , F →.id } = I_4
goto[I_0,id] =closure({F→id.}) = I_5
goto[I_1,+] = closure({ E →E+.T})
={ E →E+.T ,T →.T*F, T→.F, F→.(E), F→.id } = I_6

$goto[I_2,*] = closure(\{T \rightarrow T*.F\}) = \{ T \rightarrow T*.F, F \rightarrow .(E), F \rightarrow .id \} = I_7$
$goto[I_4,E] = closure(\{F \rightarrow (E.), E \rightarrow E.+T\}) = \{F \rightarrow (E.), E \rightarrow E.+T\} = I_8$
$goto[I_4,T] = closure(\{E \rightarrow T. , T \rightarrow T.*F\}) = \{E \rightarrow T. , T \rightarrow T.*F\} = I_2$
$goto[I_4,F] = closure(\{T \rightarrow F.\}) = \{T \rightarrow F.\} = I_3$
$goto[I_4,(] = closure(\{F \rightarrow .(E)\} = I_4$
$goto[I_4,id] = closure(\{F \rightarrow id.\}) = I_5$
$goto[I_6,T] = closure (\{E \rightarrow E+T. , T \rightarrow T.*F\}) = \{E \rightarrow E+T. , T \rightarrow T.*F\} = I_9$
$goto[I_6,F] = closure (\{T \rightarrow F.\}) = I_3$
$goto[I_6,id] = closure(\{F \rightarrow id.\}) = I_5$
$goto[I_7,F] = closure(\{T \rightarrow T*F.\}) = \{ T \rightarrow T*F.\} = I_{10}$
$goto[I_7,(] = closure(\{F \rightarrow .(E)\} = I_4$
$goto[I_7,id] = closure(\{F \rightarrow id.\}) = I_5$
$goto[I_8,)] = closure(\{F \rightarrow (E).\}) = \{ F \rightarrow (E).\} = I_{11}$
$goto[I_8,+] = closure(\{ E \rightarrow E+.T\}) = I_6$
$goto[I_9,*] = closure(\{T \rightarrow T*.F\}) = \{ T \rightarrow T*.F, F \rightarrow .(E), F \rightarrow .id \} = I_7$

The transition diagram of the goto function : DFA



Construction of SLR parsing Table

To construct the action and goto parts of a SLR parsing table , use the following algorithm.

1. Given the grammar G, construct an augmented grammar G' by introducing a production of the form S'$\rightarrow S$ where S is the start symbol of the grammar G.
2. Let I_0 = closure({S'$\rightarrow .S$ }) , starting from I_0 construct SLR (canonical collection of sets of LR(0))items for G' using closure and goto: **C$\leftarrow${I_0,...,I_n}**
3. State i is constructed from I_i . then the parsing actions for state i are defined as follows:
   a. If goto[I_i,a] =I_j, set action[i,a] = " shift j" , here a must be a termial.
   b. If I_i has a production of the form $A\rightarrow \alpha.$, then for all symbol in FOLLOW(A), set action[i,a] = " Reduce $A\rightarrow \alpha$" , here A should not be S'
   c. If [S'$\rightarrow S.$] is in I_i , then set action[i,$] = Accept.
4. If goto[I_i,A] = I_j , where A is non terminal, then set goto[i,A] = j
5. For all blank entries not defined by step 2 to 3, set error.
6. The start state s_0 corresponds to I_0 = closure({S'$\rightarrow .S$ }).

- For any conflicting actions are generated by the above rules, we say the grammar is not SLR. The algorithm fails to produce a parser in this case.
- The parsing table constructed by this algorithm is SLR(1) parsing table of G.
- An LR parser using this SLR(1) table is called SLR(1) parser and simple called as SLR parser.

Example: Construct the SLR parsing table for the following grammar.

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

Solution: The augmented grammar G' is:

$E'\rightarrow E$
$E\rightarrow E+T \mid T$
$T\rightarrow T*F \mid F$
$F\rightarrow (E) \mid id$

The canonical collection of set of LR(0) items for this grammar are:

I_0: $E'\rightarrow .E$, $E\rightarrow .E+T$, $E\rightarrow .T$, $T\rightarrow .T*F$, $T\rightarrow .F$, $F\rightarrow .(E)$, $F\rightarrow .id$

I_1: $E'\rightarrow E.$, $E\rightarrow E.+T$

I_2: $E\rightarrow T.$, $T\rightarrow T.*F$

I_3: $T\rightarrow F.$

I_4: $F\rightarrow (.E)$, $E\rightarrow .E+T$, $E\rightarrow .T$, $T\rightarrow .T*F$, $T\rightarrow .F$, $F\rightarrow .(E)$, $F\rightarrow .id$

I_5: $F\rightarrow id.$

I_6: $E\rightarrow E+.T$, $T\rightarrow .T*F$, $T\rightarrow .F$, $F\rightarrow .(E)$, $F\rightarrow .id$

I_7: $T\rightarrow T*.F$, $F\rightarrow .(E)$, $F\rightarrow .id$

I_8: $F\rightarrow (E.)$, $E\rightarrow E.+T$

I_9: $E\rightarrow E+T.$, $T\rightarrow T.*F$

I_{10}: $T\rightarrow T*F.$

I_{11}: $F\rightarrow (E).$

Now computing the entry for the SLR parsing table for action table,

Consider the set of items
For I_0:
The item F→.(E) gives to the entry for **action[0,(] = shift 4**
The item **F→.id** gives to the entry for **action [0,id] = shift 5**

The other items in I_0 yield no action. Similarly
For I_1:
action[1,$] = accept since ***E' →E.*** is in I_1
action[1,+] = shift 6
For I_2:
action[2,*] = shift 7 since T→T.*F is in I_2
From E →T. , FOLLOW(E) = { $,+,)} so action[2,$] =action[2,+]=action[2,)]=reduce E →T
For I_3 :
T → F. , so FOLLOW(T) = {*,$,+,)}
So, action[3,*] = action[3,$] = action[3,+] = action[3,)] = reduce T → F
For I_4:
action[4,( ] = shift 4
action[4,id] = shift 5
For I_5:
FOLLOW(F) = {*,$,+,)}
So, action[5,*] = action[5,$] = action[5,+] = action[5,)] = reduce F→ id

Similarly,

For I_6:
action[6,(] = shift 4
action[6,id] = shift 5
For I_7:
action[7,(] = shift 4
action[7,id] = shift 5
For I_8:
action[8,)] = shift 11
action[8,+] = shift 6
For I_9:
E →E+T. , FOLLOW(E) ={ $,+,)}
So, action[9,$] =action[9,+]=action[9,)]=reduce E →***E***+T and
action[9,*] = shift 7
For I_{10}:
T → T*F. , FOLLOW(T) = {*,$,+,)}
So, action[10,*] = action[10,$] = action[10,+] = action[10,)] = reduce T → ***T****F
For I_{11}:
F → (E). , FOLLOW(F) = {*,$,+,)}
So, action[11,*] = action[11,$] = action[11,+] = action[11,)] = reduce F→ ***(E)***

Now the action table for SLR parsing for above grammar is:

States	Terminals					
	id	+	*	(	)	$
0	shift 5			shift 4		
1		shift 6				accept
2		E →T			E →T	E →T
3		T→F	T→F		T→F	T→F
4	shift 5			shift 4		
5		F→id	F→id		F→id	F→id
6	shift 5			shift 4		
7	shift 5			shift 4		
8		shift 6			shift 11	
9		E→***E+T***	shift 7		E→***E+T***	E→***E+T***
10		T→***T*F***	T→***T*F***		T→***T*F***	T→***T*F***
11		F→***(E)***	F→***(E)***		F→***(E)***	F→***(E)***

Now **goto** table for SLR.

States→ Variable↓	**E**	**T**	**F**
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Using the above parsing table, the SLR Parser takes the following moves to parse the string ***id*id+id***

Stack	**Input**	**Action**	**Output**
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Properties of SLR grammars

Every SLR(1) grammar is unambiguous. But there exist certain unambiguous grammar that are not SLR(1). In such a grammars there exists at least one multiply defined entry action[i,a] which contains both shift directive and reduce directive.

Invalid reduction:

In SLR parser , in any state i, a reduction $A\rightarrow \alpha$ is performed on input symbol 'a' if state i contains $[A\rightarrow \alpha.]$ and 'a' is in FOLLOW(A), however not all symbols in FOLLOW(A) can be reduced in such a fashion.

For example, consider following grammar.

G:
S → L=R
S → R
L→ *R
L → id
R→ L

⇒

G' :
S'→S
S → L=R
S → R
L→ *R
L → id
R→ L

The canonical collection of the item sets of LR(0) for this grammar are

I_0: **S'**→.S
S → .L=R
S →. R
L→ . *R
L →.id
R→ . L

I_1: S'→S.

I_2: S→ L.=R
R→ L.

I_3: S →R.

I_4:
L→ *.R
R→ . L
L→ . *R
L →.id

I_5: L →id.

I_6: S → L=.R
R→ . L
L→ . *R
L →.id

I_7: L→ *R.

I_8: R→ L.

I_9: S → L=R.

Now: while construction the parsing table, consider the item set I_2:
goto[I_2,=] = I_6 this will make entry in SLR parsing table as action[2,=] = shift 6

also R→ L. is also in I_2 , so by the rule of constructing SLR parsing table , we compute
FOLLOW(R) = { =,$} this will make entry to SLR parsing table as action[2,=] = reduce R→ L.

So the entry on action table for action[2,=] is multiply defined, one is shift operation and another is reduce operation. This leads the SLR parser into shift-reduce conflict for the state 2 and input '='.

The grammar is not ambiguous but there is shift-reduce conflict. So the SLR parser is not enough powerful to remember left context to decide what action the parser should take on input '='.

LR(k) items

In order to avoid invalid reductions, the general form of an item is of the form [A→ α.β,a] i.e. extra information is put into a state by including a terminal symbol as a second component in an item. In this case, the second term a has no effect when β is not empty, but in an item of the form [A→ α., a], reduce action is performed using form [A→ α] only if the next input symbol is 'a'.

Such an item is called a LR(1) item where the input symbol 'a' is called the "lookahead" whose length is 1.

The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

Computation of closure(1) and LR(1) items.

1. *Repeat*
2. *For each item of the form [A→ α.Bβ,a] in I,*
   each production of the form B → γ in G' and
   each terminal 'b' in FIRST(βa) do
   add [B → .γ,b] to I if it is not already there.
3. *Until no more items can be added to I.*

Computation of goto[I,X] for LR(1) items

Given the set of all items of the form [A→ α.Xβ,a] in I,
goto[I,X] = closure({A→ αX.β,a}).

Computation of LR(1) DFA

1. *Start with C = {colsure({S'→ .S,$}) where S is start symbol.*
2. *Repeat*
3. *For each set of items I in C and each grammar symbol X such that goto[I,X] is not already in C, do,*
   Add goto[I,X] to C.
4. *Until no more sets of items can be added to C.*

Example: Compute the LR(1) collection of items from the following grammar.

S→ *CC*

C→*cC | d*

Solution: The augmented grammar is

S'→S

S→ *CC*

C→*cC | d*

I_0: closure({(S' → • S, $)})
={ (S' → • S, $)
(S → • C C, $)
(C → • c C, c/d)
(C → • d, c/d)}

I_1: goto(I_0, S)
= (S' → S • , $)

I_2: goto(I_0, C) =
(S → C • C, $)
(C → • c C, $)
(C → • d, $)

I_3: goto(I_0, c) =
(C → c • C, c/d)
(C → • c C, c/d)
(C → • d, c/d)

I_4: goto(I_0, d) =
(C → d •, c/d)

I_5: goto(I_2, C) =
(S → C C •, $)

I_6: goto(I_2, c) =
(C → c • C, $)
(C → • c C, $)
(C → • d, $)

I_7: goto(I_2, d) =
(C → d •, $)

I_8: goto(I_3, C) =
(C → c C •, c/d)

goto(I_3, c) = I_3

goto(I_3, d) = I_4

I_9: goto(I_6, C) =
(C → c C •, $)

goto(I_6, c) = I_6

: goto(I_6, d) = I_7

Note: In first case [S' → • S, $] is of the form *[A→ α.Bβ,a] where β is empty and a ='$'. It has to be added the item [B → .γ,b] for each terminal 'b' in FIRST(βa) which is equal to '$'*
So add S → • C C, $)
Similarly for case S → • C C, $) , *(βa) = (C$) and FIRST(C$) ={c, d} so it has to be added the items*

C→•*cC,c/d*
C→•*d,c/d*

The Core of LR(1) Items

The core of a set of LR(1) Items is the set of their first components (i.e., LR(0) items). For example the core of the set of LR(1) items

{ (C → c • C, c/d),
(C → • c C, c/d),
(C → • d, c/d) }

is

{ C → c • C,
C → • c C,
C → • d
}

Construction of the LR(1) parsing table

To construct the action and goto parts of a LR(1) parsing table , use the following algorithm.

1. Given the grammar G, construct an augmented grammar G' by introducing a production of the form $S' \rightarrow S$ where S is the start symbol of the grammar G.
2. Let I_0 = closure({S' $\rightarrow$ .S,$ }) , starting from I_0 construct LR(1) (canonical collection of sets of LR(1))items for G' using closure and goto: **C←{I_0,...,I_n}**
3. State i is constructed from I_i . then the parsing actions for state i are defined as follows:
   a. If goto[I_i,a] =I_j, set action[i,a] = " shift j" , here a must be a terminal.
   b. If I_i has a production of the form [**A** $\rightarrow$ **α. ,a] ,** set action[i,a] = " Reduce **A** $\rightarrow$ **α**" , here A should not be S'
   c. If [S' $\rightarrow$ *S.,$]* is in I_i , then set action[i,$] = Accept.
4. If goto[I_i,A] = I_j , where A is non terminal, then set goto[i,A] = j
5. For all blank entries not defined by step 2 to 3, set error.
6. The start state s_0 corresponds to I_0 = closure({S' $\rightarrow$ *.S,$ }).*

Now the parsing table for the grammar given above as

1. S' $\rightarrow$ *S*
2. S $\rightarrow$ *CC*
3. C $\rightarrow$ *cC*
4. C $\rightarrow$ *d*

will be as:

Terminals/ States	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	R3	R3			
5			R1		
6	s6	s7			9
7			R3		
8	R2	R2			
9			R2		

Here **R2** means Reduce by Production 2 of grammar above
i.e. Reduce S $\rightarrow$ *CC* and so on.
s_i means shift i i.e. s3 refers as "shift 3"

LALR Grammar

LALR(Lookahead LR) grammars are midway in complexity between SLR and canonical LR(most complex) grammars. Ther perform a "generalization" over canonical LR itemsets.

A typical programming language generates thousands of states for canonical LR parser while they generate only hundreds of states for SLR and LALR. So it is much easier and economical to construct the SLR and LALR parser.

Union operation on items.

- Given an item of the form $[A \rightarrow \alpha.B\beta,a]$, the first part of the item $A \rightarrow \alpha.B\beta$ *is called the "core" of the item.*
- Given two states of the form $I_i = A \rightarrow \alpha.,a$ and $I_j = A \rightarrow \alpha.,b$ the core of I_i and I_j is same only the difference is the second part of the item. So the union of these two items is defined as

  $I_{ij} = \{[A \rightarrow \alpha.,a/b]\}$.
- The I_{ij} will perform the reduce operation on seeing either 'a' or 'b' on input buffer. The state machine resulting from the above union operation has one less state.
- If I_i and I_j have more than one items, then the set of all core elements in I_i should be the same as the set of all core elements in I_j for the union operation to be possible.
- Union operation do not create any new shift-reduce conflicts but can create new reduce-reduce conflicts. Shift operation only depends on the core and not on the next input symbol.

Construction of the LALR parsing table

To construct the action and goto parts of aLALR parsing table , use the following algorithm.

1. Given the grammar G, construct an augmented grammar G' by introducing a production of the form $S' \rightarrow S$ where S is the start symbol of the grammar G.
2. For each core present; find all sets having that same core; replace those sets having same cores with a single set which is their union. $C=\{I_0,...,I_n\}$ ➔ $C'=\{J_1,...,J_m\}$ where $m \leq n$
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.

   If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

   ➔ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
4. So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.

   If no conflict is introduced, the grammar is LALR(1) grammar. The above algorithm is inefficient since it constructs the entire canonical DFA before generating the LALR parsing table.

Example:
Compute the LR(1) collection of items from the following grammar.
$S \rightarrow CC$
$C \rightarrow cC \mid d$

Solution: The augmented grammar is

S'→S

S→*CC*

C→*cC* | *d*

The canonical LR(1) items computed from this grammar are { $I_0, I_1, ..., I_n$} as computed in previous LR(1) parsing table.

In the collection of LR(1) items, I_3 and I_6, I_4 and I_7, I_8 and I_9 have same core items respectively. So performing union operations, the items for LALR will be as

I_0: closure({(S' → • S, $)})
={ (S' → • S, $)
(S → • C C, $)
(C → • c C, c/d)
(C → • d, c/d)}

I_1: goto(I_0, S)
= (S' → S • , $)

I_2: goto(I_0, C) =
(S → C • C, $)
(C → • c C, $)
(C → • d, $)

I_{36}: goto(I_0, c) =
(C → c • C, c/d/$)
(C → • c C, c/d/$)
(C → • d, c/d/$)

I_{47}: goto(I_0, d) =
(C → d •, c/d/$)

I_5: goto(I_2, C) =
(S → C C •, $)

I_{89}: goto(I_3, C) =
(C → c C •, c/d/$)

Now,

goto[I_{36},C] = I_{89}, since goto[I_3,C] = I_8 in original set of LR(1) items. Similarly goto[I_2,c] = I_{36}, since goto[I_2,c] = I_6 in original LR(1) items and so on.

So the LALR parsing table for the grammar has 3 less states as

Terminals/ States	**action**			**goto**	
	c	**d**	**$**	**S**	**C**
0	**s36**	**s4**		**1**	**2**
1			**accept**		
2	**s36**	**s7**			**5**
36	**s36**	**s4**			**89**
47	**R3**	**R3**	**R3**		
5			**R1**		
89	**R2**	**R2**	**R2**		

Now for input ccd the parser takes the following steps in LR(1) parser

Stack	inputbuff	action	output
$0	ccd$	shift 3	
$0c3	cd$	shift 3	
$0c3c3	d$	shift 4	
$0c3c3d4	$	Error	

In LALR using above table,

Stack	inputbuff	action	output
$0	ccd$	shift 36	
$0c36	cd$	shift 36	
$0c36c36	d$	shift 47	
$0c36c36d47	$	reduce C→d	C→d
$0c36c36C89	$	reduce C→cC	C→cC
$0c36C89	$	reduce C→cC	C→cC
$0C2	$	Error	

So for both LR(1) and LALR parser leads to an Error for string **ccd.**
Now consider for string *cdd*

LR(1) parser				**LALR Parser**			
Stack	**Input**	**Action**	**output**	**Stack**	**Input**	**Action**	**output**
$0	**cdd$**	**shift 3**		**$0**	**cdd$**	**shift 36**	
$0c3	**dd$**	**shift 4**		**$0c36**	**dd$**	**shift 47**	
$0c3d4	**d$**	Red. C→d	C→d	**$0c36d47**	**d$**	Red . C→d	C→d
$0c3C8	**d$**	Red. C→cC	C→cC	**$0c36C89**	**d$**	Red. C→cC	C→cC
$0C2	**d$**	**shift 7**		**$0C2**	**d$**	**shift 47**	
$0C2d7	**$**	Red . C→d	C→d	**$0C2d47**	**$**	Red . C→d	C→d
$0C2C	**$**	Red. S→CC	S→CC	**$0C2C5**	**$**	Red. S→CC	S→CC
$0S1	**$**	Acctpt		**$0S1**	**$**	**Accept**	

For the parsing of valid string of grammar we saw that both LR(1) and LALR take same actions for parsing. But for invalid string like ccd, the LR(1) parser leads to error after some shift action where LALR proceeds some reductions after LR parser has detected an error. But finally, LALR also discovered the error.

Kernel and non-kernel items

In order to devise a more efficient way of building LALR parsing tables, we define kernel and non-kernel items. Those items that are either the initial item [S'→.S,$] or the items that have somewhere dot(.) other than the beginning of the right side are called the kernel items. No item generated by a goto has dot at the left end of production. Items that are generated by closure over kernel items hav a dot at the beginning of production. These items are called non-kernel items.