[Syntax Analysis]

Compiler Design and Construction (CSc 352)

Compiled By

Bikash Balami

Central Department of Computer Science and Information Technology (CDCSIT)

Tribhuvan University, Kirtipur

Kathmandu, Nepal

Syntax Analysis

All programming languages have certain syntactic structures. We need to verify that the source code written for a language is syntactically valid. The validity of the syntax is checked by the syntax analysis. Syntaxes are represented using context free grammar (CFG), or Backus Naur Form (BNF). Parsing is the act of performing syntax analysis to verify an input program's compliance with the source language. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. A by-product of this process is typically a tree that represents the structure of the program. Parsing is the act of checking whether a grammar "accepts" an input text as valid (according to the grammar rules). It determines the exact correspondence between the text and the rules of given grammar.

Role of the Parser

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It has to report any syntax errors if occurs.

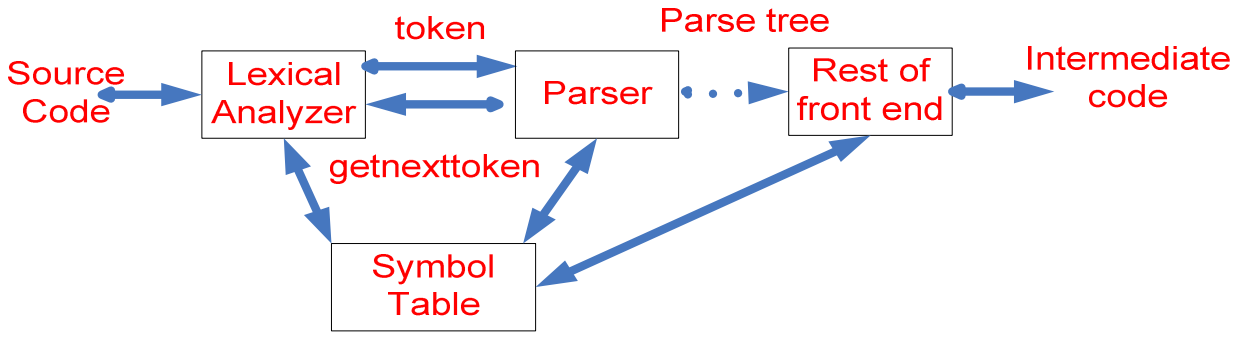


Figure :- Role of Parser in a compiler model

The tasks of parser can be expressed as

1. Analyzes the context free syntax
2. Generates the parse tree
3. Provides the mechanism for context sensitive analysis
4. Determine the errors and tries to handle them

Types of Parser

1. **Universal Parser**
    - Can parse any kind of grammars
    - Inefficient for compiler construction due to inefficient to use in production compilers
    - E.g. Cocke-Younger-Kasami algorithm and Earley's algorithm
2. **Top Down Parser**
    - Builds parse tree form the top(root) to the bottom(leaves)
    - Easy to build
    - Works on subset of grammars
    - Eg:- LL grammars
3. **Bottom Up Parser**
    - Builds parse tree from bottom(leaves) to up(root)
    - Difficult to build
    - Works on subset of grammars
    - Eg:- LR grammars

Error Recovery Strategies (Error Handling)

If a compiler had to process only correct programs, its design and implementation would be simplified. But it would be like, the programmer can frequently write incorrect programs, and a good compiler should assist the programmer in identifying and correcting the errors. Every phases of the compiler is prone to errors and more common source of errors are two phases syntax analysis and semantic analysis. For example, syntactic error such as *an arithmetic expression with unbalanced parenthesis*. If the error occurs the process should not terminate instead report the error and advance.

Error Recovery Techniques

- Panic mode recovery
- Phrase level recovery
- Error productions
- Global correction

Panic Mode Recovery

- Simplest method on which upon discovering the error parser discards the input one at a time, until one of a designated set of some synchronizing token is found, such as semicolon
- May skip errors if there are more than one error in the sentence
- No infinite loop

Phrase Level Recovery

- Error discovery lead to the local correction (choice is arbitrary) like by replacing the prefix of the remaining input by the string that will allow the process to continue
- A typical local correction may be like replacing a comma by semicolon, delete an extraneous semicolon or inserting missing semicolon
- May go in infinite loop if the insertion is done so that it is ahead of the current input symbol always
- Some time it is difficult to cope with errors if cause of error is occurred before the detection

Error Productions

- If we have good knowledge of errors then we can augment the grammar that produces errors
- We can then use the grammar augmented by these error production to construct a parser.
- If an error production is used by the parser, we can generate appropriate error diagnostics to indicate the erroneous construct that has been recognized in the input

Global Correction

- To make few changes as possible in processing input some algorithm determines input string X similar to Y but with no errors and the number of changes are small
- Too costly with respect to time and space, so of theoretical only

Context Free Grammar

Generally most of the programming languages have recursive structures that can be defined by Context free grammar (CFG). CFG can be defined as 4 – tuple (V, T, P, S), where

- V → finite set of Variables or Non-terminals that are used to define the grammar denoting the combination of terminal or no-terminals or both
- T → Terminals, the basic symbols of the sentences. Terminals are indivisible units
- P → Production rule that defines the combination of terminals or non-terminals or both for particular non-terminal
- S → It is the special non-terminal symbol called start symbol

Example, to define a palindrome string over binary string

S → 0S0 | 1S1

S → 0 | 1

Example, grammar to define an infix expression

E → E A E | (E) | -E | id

A → + | - | * | / | ^

E and A are non terminals with E as start symbol. Remaining are non terminals.

CFG Terms

- **Terminal**: An atomic (indivisible) building block.
- **Non-terminal**: A building block of the CFG that are made of other terminals and/or non-terminals.
- **Production rule**: Rules that define how a non-terminal is constructed from other terminals and non-terminals.
- **Start symbol**: The start symbol is a non-terminal that defines the starting point for the definition of the entire grammar.

CFG Notational Convenience

- Lowercase letters, symbols (like operators), bold string are used for denoting terminals. Eg :- a, b, c, 0, 1 $\in$ T

- Uppercase letters, italicized strings are used for denoting non-terminals. Eg :- A, S, B, C $\in$ V
- Lowercase Greek letters (α, β, γ, δ) are used to denote the terminal, non-terminal or combination of both
- Production of the form A → a | b, is read as "*A produces a or b*"

Derivations

There are several ways to view the process by which a grammar defines a language. String of non-terminals and terminals is called sentential form, whereas string of terminals is called sentences. The main purpose of the grammar is to define the language and the verification of the sentence defined by the grammar is done using the production rule to obtain the particular sentence by expansion of the non-terminals is called **derivation**. During the derivation, if we always expand the leftmost non-terminal first then it is called leftmost derivation, defining similarly the right most derivation used rightmost non-terminal first. The term $\alpha \Rightarrow \beta$ is used to denote that β can be derived from α. Similarly, $\alpha \Longrightarrow * \beta$ denotes derivation using zero or more rules and $\alpha \Longrightarrow^{+} \beta$ denoted derivation using one or more rules. L(G) is the language of G *(the language generated by G*) which is a set of sentences. A string of L(G) is a string of terminal symbols of G. i.e. If S is a start symbol of G then, *w* is a sentence of L(G) if and only if $S \Longrightarrow w$, where *w* is a string of terminals of G. If G is a context free grammar, the L(G) is a context free language. Two grammars are equivalent if they produce the same language.

Example

Consider a grammar G (V, T, P, S), where

V → {E}

T → {+, *, -, (,), id}

P → {E → E + E, E → E * E, E → (E), E → -E, E → id}

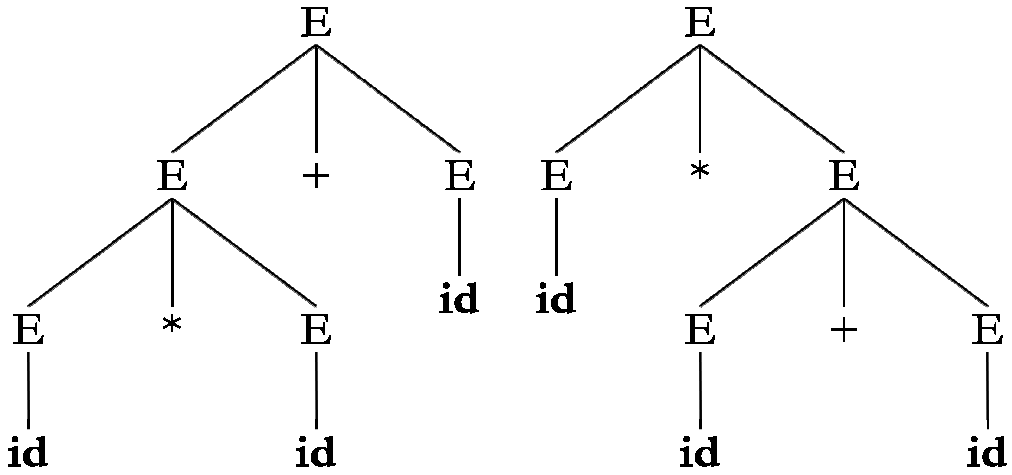
S → {E}

*For, id * id + id*

$E \Longrightarrow E * E \Longrightarrow id * E \Longrightarrow id * E + E \Longrightarrow id * id + E \Longrightarrow id * id + id$

Parse Tree

The pictorial representation of the derivation can be depicted using the parse tree. In parse tree internal nodes represent non-terminals and the leaves represent terminals. Take the grammar E → E+E | E*E | **id**, parse trees (correspond to leftmost derivation) for the sentence **id** * **id** + **id** are



Ambiguity

A grammar is said to be ambiguous if it can produce a sentence in more than one way. If there are more than one parse tree for the sentence or derivation (left or right) with respect to the given grammar then the grammar is said to be ambiguous.

Example:-

E → E + E

E → E * E

E → id

Consider the string id + id * id

Derivation 1

E E + E id + E id + E * E id + id * E id + id * id

Derivation 2

E E * E E * id E + E * id E + id * id id + id * id

(see above parse tree also resembles ambiguity for string id * id + id)

Parsing

Given a stream of input tokens, parsing involves the process of "reducing" them to a non-terminal. The input string is said to represent the non-terminal it was reduced to.

Types of Parsing

- **Top down parsing**
  - Parsing involves generating the string from the first non – terminal and repeatedly applying production rules.
- **Bottom up parsing**
  - Parsing involves repeatedly rewriting the input string until it ends up in the first non – terminal of the grammar.

Top Down Parsing

Top down parsing tries to derive the input string using leftmost derivation i.e. starting at the start non-terminal symbol the use of the production rules leads to the input string. It includes

- **Recursive Descent parsing**
  - Backtracking is needed, i.e. if a choice of production rule does not work, we have to backtrack to check for next production rule
  - Not widely used
  - Not efficient
- **Non-recursive Predictive Parsing**
  - No backtracking
  - Efficient
  - Use LL (Left – to – right scanning, Left – to – right derivation) method
  - Needs a special form of grammars (LL(1) grammar)
  - Non – recursive (Table Driven) predictive parser is also known as LL(1) parser

Recursive Descent Parsing

Start with the starting non-terminal and use the first production, verify with input and if there is no match backtrack and apply another rule.

Rules

1. Use two pointers *iptr* for pointing the input symbol to be read and *optr* for pointing the symbol of output string that is initially start symbol S.
2. If the symbol pointed by *optr* is non-terminal use the first production rule for expansion.
3. While the symbol pointed by *iptr* and *optr* is same increment the both pointers.
4. The loop at the above step terminates when
    a. A non-terminal at the output or (case A)
    b. The end of input (case B)
    c. Un matching terminal symbol pointed by *iptr* and *optr* is seen (case C)
5. If (A) is true, expand the non-terminal using the first rule and goto step 2
6. If (B) is true, terminate with success
7. If (C) is true, decrement both the pointers to the place of last non-terminal expansion and use the next production rule for non-terminal
8. If there is no more production and (B) is not true report error

Example

Consider a grammar

S → cAd

A → ab | a

Input string is "*cad*"

Input	**Ouput**	**Rules Fired**
(iptr)cad	(optr)S	[Rule 1, Try S → cAd]
(iptr)cad	(optr)cAd	[Rule 3, Match c]
c(iptr)ad	c(optr)Ad	[Rule 5, Try A → ab]
c(iptr)ad	c(optr)abd	[Rule 3, Match a]
ca(iptr)d	ca(optr)bd	[Rule 7, dead end, backtrack]
c(iptr)ad	c(optr)Ad	[Rule 5, Try A → a]
c(iptr)ad	c(optr)ad	[Rule 3, Match a]
ca(iptr)d	ca(optr)d	[Rule 3, Match d]
cad(iptr)	cad(optr)	

Terminate with success (Rule 6)

Left Recursion

The grammar with recursive non-terminal at the left of the production is called left recursive grammar. A grammar is left recursive if it has a non – terminal "*A*" such that there is a derivation,

$A \Longrightarrow A\,\alpha$ for some string α

Example

E → E+E | E*E | **id**

Left recursion causes recursive descent parser to go into infinite loop. Top down parsing techniques cannot handle left – recursive grammars. So, we have to convert our left recursive grammar which is not left recursive. The left recursion may appear in a single derivation called immediate left recursion, or may appear in more than one step of the derivation.

Example of non immediate left recursion

S → Aab | c

A → Sc |c

This grammar is not immediately left recursive, but it is still left recursive. i.e. $S \Longrightarrow Aab \Longrightarrow Scab$ or, $A \Longrightarrow Sc \Longrightarrow Aabc$ causes left recursion.

Removing Left Recursion

If we have the grammar of the form A → Aα | β, the rule for removing the left recursion is

A → βA'

A' → αA' | ε.

This is the equivalent non-recursive grammar.

Any immediate left-recursion can be eliminated by generalizing the above. Any production of the form:

$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$.

we have,

$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_m A'$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' | \varepsilon$

Algorithm

Input → Grammar G

Output → Equivalent grammar with no left recursion

Arrange non terminals in some order $A_1, A_2, \ldots\ldots, A_n$

for i=1 to n do

    for j=1 to i-1 do

        replace each production

        $A_i \rightarrow A_j\gamma$ by

        $A_i \rightarrow \alpha_1\gamma|\ldots..|\ \alpha_k\ \gamma$, where $A_j \rightarrow \alpha_1|\ldots..|\ \alpha_k$

    end do

eliminate left recursions among A_i productions

end do

Example

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

Let us order the non terminals as S , A

For S:

- Don't enter the inner loop
- Also there is no immediate left recursion to remove in S as outer loop says

For A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
- Then we have, $A \rightarrow Ac \mid Aad \mid bd \mid f$
- Now, remove the immediate left recursions
- $A \rightarrow bdA' \mid fA'$
- $A' \rightarrow cA' \mid adA' \mid \varepsilon$

So, the resulting equivalent grammar with no left recursion is

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Left Factoring

Backtracking is expensive operation so if we are able to predict the unique production rule while parsing then we are saving a lot of time resource. Left factoring is the approach of transforming the grammar that has two or more production rules with same initial substrings to the one that has not so that it will be useful for predictive parsing. For the grammar of the form

$A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n$

left factoring yields

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1|\beta_2|\ldots|\beta_n$

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 backtracking. It also suffers from the left-recursion problem. The recursive descent parser can work efficiently if there is no need of backtracking. For this purpose we can use the left recursion removed left factored grammar so that we can predict the production to use hence the name predictive parsing. For this purpose we need to be able to find the terminals that can come at the first when we expand the non-terminal. A predictive parser tries to predict which production produces the least chances of a backtracking and infinite looping. A predictive parser is characterized by its ability to choose the production to apply solely on the basis of the next input symbol and the current non terminal being processed. To enable this, the grammar must take a particular form. We call such a grammar LL(1). The first "L" means we scan the input from left to right; the second "L" means we create a leftmost derivation; and the 1 means one input symbol of lookahead. Informally, an LL(1) has no left-recursive productions and has been left-factored.

Example

Stmt → if …….|
while ……|
begin ……|
for…..

Then, when are trying to write the non terminal *stmt*, then if the next token is *while*, then we use the second production rule, instead of using first production as Recursive Descent Parser does.

Types of Predictive Parsing

- Recursive (Recursive Predictive Parsing)
- Non Recursive (Table Driven Parsing)

Non Recursive Predictive Parsing

It is possible to build a non recursive predictive parser by maintaining a stack explicitly rather than implicitly through recursive calls.

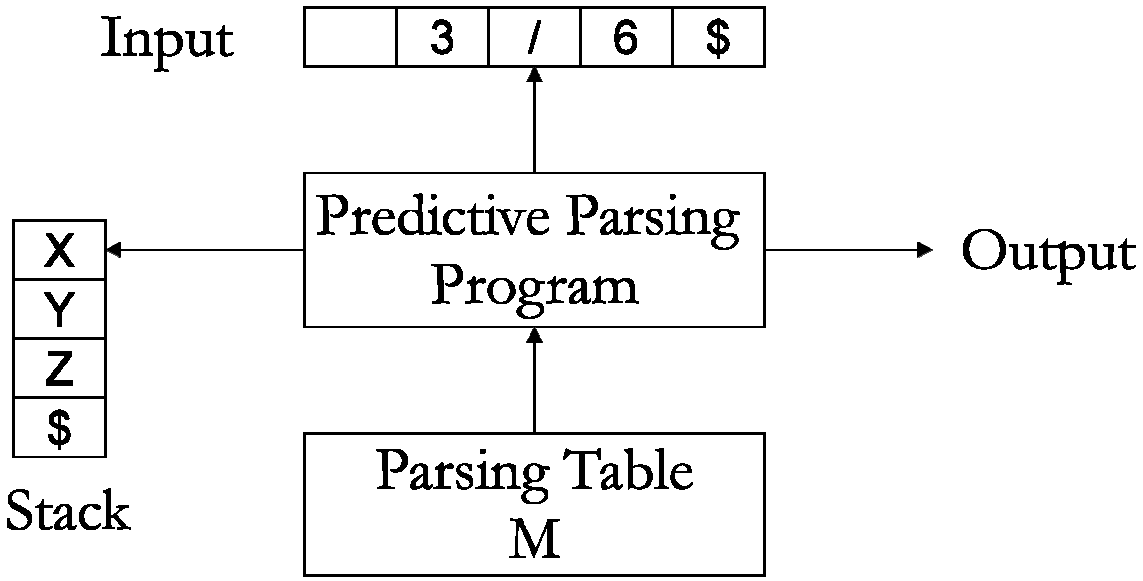


Figure :- Non recursive predictive parsing

Non recursive predictive parsing is a table driven parser. The table driven predictive parser has stack, input buffer and parsing table and output stream. The input buffer contains the sentence to be parsed and at the end $ as an end marker. The stack contains symbols of the grammar. Initially stack contains start symbol on top of $.When the stack is emptied (i.e. only $ left in the stack), the parsing is completed. Parsing table is two dimensional array containing the information about the production to be used on seeing the terminal at input and non-terminal at stack. Each entry in the parsing table holds the production rule. Each row holds the terminal or $, and each column holds non terminal symbols.

Algorithm

Input → a string *w*

Output → if *w* is in L(G), a leftmost derivation of w, otherwise *error*

set ip to point the first symbol of input stream and set the stack $S where S is the start symbol (initially top of stack).

do

let X be top of the stack and a be the symbol pointed by ip.

if X is a terminal or $ then

if X = a then pop X from the stack and forward ip else error()

else

if M[X, a] = X → $Y_1, Y_2, \ldots, Y_n$. then

pop X from the stack

push $Y_n, Y_{n-1}, \ldots, Y_1$, with Y_1 at the top

output the production $X \rightarrow Y_1, Y_2, \ldots, Y_n$.

else

error()

until X = $.

Figure :- Algorithm for Non Recursive Predictive Parsing

Example

Consider the grammar

S → aBa

B → bB | ε

Parsing table be (we will see how to construct parsing table later)

	a	b	$
S	S → aBa		
B	B → ε	B → bB	

Table :- LL(1) Parsing table

Given string *w* = abba

Stack	**Input**	**Output**
$S	abba$	S → aBa
$aBa	abba$	
$aB	bba$	B → bB
$aBb	bba$	

$aB	ba$	B → bB
$aBb	ba$	
$aB	a$	B → ε
$a	a$	
$	$	Accept and successful completion

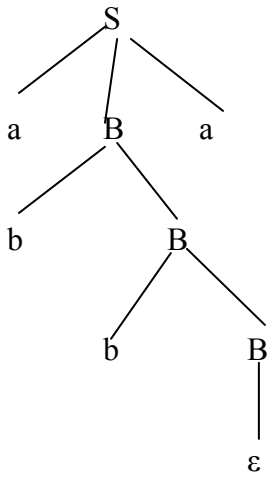
Output :-

S → aBa, B → bB, B → bB, B → ε

So, the left most derivation is

S ⟹ aBa ⟹ abBa ⟹ abbBa ⟹ abba

And the parse tree will be



Constructing LL(1) Parsing Table

The parse table construction requires two functions: FIRST and FOLLOW. For a string of grammar symbols α, FIRST(α) is the set of terminals that begin all possible strings derived from α. If α ⟹* ε, then ε is also in FIRST(α). FOLLOW(A) for non terminal A is the set of terminals that can appear immediately to the right of A in some sentential form. If A can be the last symbol in a sentential form, then $ is also in FOLLOW(A).

Computation of FIRST

Rules:

- FIRST(α) = {the set of terminals that begin all strings derived from α}
- If "*a*" is a terminal symbol them FIRST(a) = {a}
- FIRST(ε) = {ε}

- FIRST(A) = $\cup_{A\to\alpha}$ FIRST(α) for $A\to\alpha \in P$
- To calculate FIRST(α) where $\alpha \to X_1X_2\ldots\ldots X_n$, do the following
    - If X_1 is a terminal, add X_1 to FIRST(α) and finished
    - Else X_1 is a non terminal, so add First(X_1) - ε to FIRST(α)
    - If X_1 is a nullable non terminal, i.e., $X_1 =>^* \varepsilon$, add FIRST(X_2) - ε to FIRST(α) , Furthermore, if X_2 can also go on to ε, then add FIRST(X_3)- ε and so on, through all X_n until the first non nullable symbol is encountered.
    - If $X_1X_2...X_n =>^* \varepsilon$, add ε to the first set

Computation of FOLLOW

FOLLOW(A) = { the set of terminals that can immediately follow non terminal A }

Rules:

- For every production $B \to \alpha A \beta$, where α and β are any string of grammar symbols and A is non terminal, then everything in FIRST(β) except ε is placed on FOLLOW(A)
- For every production $B \to \alpha A$, or a production $B \to \alpha A \beta$, FIRST(β) contains ε (i.e. β is nullable), then everything in FOLLOW(B) is added to FOLLOW(A)

Example

$R \to id\ S \mid (R)S$

$S \to +RS \mid .RS \mid *S \mid \varepsilon$

FIRST(R) = { id, (}

FIRST(S) = { +, ., *, ε }

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

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

Example

Here is a complete example of first and follow set computation, starting with this grammar:

$S \to AB$

$A \to Ca \mid \varepsilon$

$B \to BaAC \mid c$

$C \to b \mid \varepsilon$

Notice we have a left-recursive production that must be fixed if we are to use LL(1) parsing:

$B \rightarrow BaAC \mid c$ becomes $B \rightarrow cB'$

$B' \rightarrow aACB' \mid \varepsilon$

Now the new grammar is:

$S \rightarrow AB$

$A \rightarrow Ca \mid \varepsilon$

$B \rightarrow cB'$

$B' \rightarrow aACB' \mid \varepsilon$

$C \rightarrow b \mid \varepsilon$

It helps to first compute the nullable set (i.e., those nonterminals X that $X \Longrightarrow^* \varepsilon$), since you need to refer to the nullable status of various nonterminals when computing the first and follow sets:

nullable(G) = {A B' C}

The first sets for each non terminal are:

First(C) = {b. ε}

First(B') = {a. ε}

First(B) = {c}

First(A) = {b, a, ε}

Start with First(C) - ε, add a (since C is nullable) and ε (since A itself is nullable)

First(S) = {b a c}

Start with First(A) - ε, add First(B) (since A is nullable). We don't add ε (since S itself is not-nullable— A can go away, but B cannot)

It is usually convenient to compute the first sets for the nonterminals that appear toward the bottom of the parse tree and work your way upward since the nonterminals toward the top may need to incorporate the first sets of the terminals that appear beneath them in the tree.

To compute the follow sets, take each nonterminal and go through all the right-side productions that the nonterminal is in, matching to the steps given earlier:

Follow(S) = {$}

S doesn't appear in the right hand side of any productions. We put $ in the follow set because S is the start symbol.

Follow(B) = {$}

B appears on the right hand side of the $S \rightarrow AB$ production. Its follow set is the same as S.

Follow(B') = {$}

B' appears on the right hand side of two productions. The B' → aACB' production tells us its follow set includes the follow set of B', which is tautological. From B → cB', we learn its follow set is the same as B.

Follow(C) = {a, $}

C appears in the right hand side of two productions. The production A → Ca tells us a is in the follow set. From B' → aACB' , we add the First(B') which is just a again. Because B' is nullable, we must also add Follow(B') which is $.

Follow(A) = {c, b, a, $}

A appears in the right hand side of two productions. From S → AB we add First(B) which is just c. B is not nullable. From B' → aACB' , we add First(C) which is b. Since C is nullable, so we also include First(B') which is a. B' is also nullable, so we include Follow(B') which adds $

Construction of LL(1) Parsing Table

Algorithm

Input → LL(1) grammar G

Output → parsing table M

For each production A → α in grammar G do

    For each a ∈ FIRST(α) do

        Add A → α to M[A, a]

    End do

    If ε ∈ FIRST(α) then

        For each b ∈ FOLLOW(A) do

            Add A → α to M[A, b]

        End do

    End if

End do

Mark each undefined entry in M error

Note: If the grammar is left recursive or ambiguous, then M will have at least one multiply-defined entry

Example

Consider the grammar

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

FIRST(F) ={(, id}

FIRST(T’) ={*, ε }

FIRST(T) ={(, id}

FIRST(E’) ={+, ε }

FIRST(E) ={(, id}

FIRST(TE’) = {(, id}

FIRST(+TE’) = {+}

FIRST(ε) = { ε }

FIRST(FT’) = {(, id}

FIRST(*FT’) = {*}

FIRST(ε) = { ε }

FIRST((E)) = {(}

FIRST(id) = {id}

FOLLOW(E) ={$,)}

FOLLOW(E’) ={$,)}

FOLLOW(T) ={+, $,)}

FOLLOW(T’) ={+, $,)}

FOLLOW(F) ={+, *, $,)}

Now do for every production

E → TE’	FIRST(TE’) = {(, id}	E → TE’ into M[E, (] and M[E, id]
E’ → +TE’	FIRST(+TE’) = {+}	E’ → +TE’ into M[E’, +]
E’ → ε	FIRST(ε) = { ε } but since ε in FIRST(ε) and FOLLOW(E’) = {$,)}	E’ → ε into M[E’, $] and M[E’,)]
T → FT’	FIRST(FT’) = {(, id}	T → FT’ into M[T, (] and M[T, id]
T’ → *FT’	FIRST(*FT’) = {*}	T’ → *FT’ into M[T’, *]
T’ → ε	FIRST(ε) = { ε } but since ε in FIRST(ε) and FOLLOW(T’) = {$,), +}	T’ → ε into M[T’, $], M[T’,)] and M[T’, +]
F → (E)	FIRST((E)) = {(}	F → (E) into M[F, (]
F → id	FIRST(id) = {id}	F → id into M[F, id]

Now, the final table looks like

NT	**Input Symbols**					
	id	*	+	(	)	$
E	E → TE’			E → TE’		
E’			E’ → +TE’		E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → *FT’	T’ → ε		T’ → ε	T’ → ε
F	F → id			F → (E)		

Example: id + id * id

Stack	**Input**	**Output**
$E	id + id * id $	E → TE’ [see above M[E, id] = E → TE’]
$E’T	id + id * id $	T → FT’
$E’T’F	id + id * id $	F → id
$E’T’id	id + id * id $	Match id, pop
$E’T’	+ id * id $	T’ → ε

$E’	+ id * id $	E’ → +TE’
$E’T+	+ id * id $	Match +, pop
$E’T	id * id $	T → FT’
$E’T’F	id * id $	F → id
$E’T’id	id * id $	Match id, pop
$E’T’	* id $	T’ → *FT’
$E’T’F*	* id $	Match *, pop
$E’T’F	id $	F → id
$E’T’id	id $	Match id, pop
$E’T’	$	T’ → ε
$E’	$	E’ → ε
$	$	

LL(1) Grammars

The predictive top-down techniques (either recursive-descent or table-driven) require a grammar that is LL(1). One fully-general way to determine if a grammar is LL(1) is to build the table and see if you have conflicts. 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 the second “L” corresponds to left-most derivations. 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 by which multiply-defined entries can be made single-valued without affecting the language recognized by a grammar. (i.e. There are no general rules to convert a non LL(1) grammar into a LL(1) grammar).

Properties of LL(1) Grammar

- No Ambiguity
- No Recursion
- 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
    - For any terminal a, if $a \in \text{FIRST}(\alpha)$ then $a \notin \text{FIRST}(\beta)$
    - Either $\alpha \Longrightarrow^* \varepsilon$ or $\beta \Longrightarrow^* \varepsilon$, but nor both
    - If $\beta \Longrightarrow^* \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A)

Examples

Grammar	**Not LL(1) Because**
$S \rightarrow S\ a \mid a$	Left Recursive
$S \rightarrow a\ S \mid a$	$\text{FIRST}(a\ S) \cap \text{FIRST}(a) \neq \varnothing$
$S \rightarrow a\ R \mid \varepsilon,\ R \rightarrow S \mid \varepsilon$	For R: $S \Rightarrow^* \varepsilon$ and $\varepsilon \Rightarrow^* \varepsilon$
$S \rightarrow a\ R\ a,\ R \rightarrow S \mid \varepsilon$	For R: $\text{FIRST}(S) \cap \text{FOLLOW}(R) \neq \varnothing$

Error Recovery in Predictive Parsing

An error may occur in the predictive parsing due to following reasons

- If the terminal symbol on the top of the stack does not match with the current input symbol.
- If the top of the stack is a non – terminal A, the current input symbol is *a*, and the entry for M[A, *a*] in parsing table is empty.

What should the parser do in an errors case?

- A parser should try to determine that an error has occurred as soon as possible. Waiting too long before declaring an error can cause the parser to lose the actual location of the error.
- A suitable and comprehensive message should be reported. “Missing semicolon on line 36” is helpful, “unable to shift in state 425” is not.

- After an error has occurred, the parser must pick a reasonable place to resume the parse. Rather than giving up at the first problem, a parser should always try to parse as much of the code as possible in order to find as many real errors as possible during a single run.
- A parser should avoid cascading errors, which is when one error generates a lengthy sequence of spurious error messages.
- It should be recovered from that error case, and it should be able to continue the parsing with the rest of the input.

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 the root (the top). The process of replacing a substring by a non-terminal in bottom - up parsing is called reduction. Along the way, a bottom-up parser searches for substrings of the working string that match the right side of some production. When it finds such a substring, it reduces it, i.e., substitutes the left side non - terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

Special Cases

- Shift Reduce Parsing
- Operator Precedence Parsing (Not included in our syllabus)
- LR Parsing

Example

Consider the grammar

S → aABe

A → Abc | b

B → d

Now, the sentence *abbcde* can be reduced to S as follows

- abbcde
- aAbcde (replacing b by using A → b)
- aAde (replacing Abc by using A → Abc)
- aABe (replacing d by using B → d)
- S

A substring that can be replaced by a non – terminal when it matches its right sentential form is called **handle**. If the grammar is unambiguous, then every right – sentential form of the grammar has exactly one handle.

Shift – Reduce Parsing

A shift reduce parser tries to reduce the given input string into the starting symbol. At each reduction step, a substring of the input matching to the right side of a production rule is replaced by non – terminal at the left side of that production rule. If the substring is chosen correctly, the rightmost derivation of that string is created in reverse order. For example in above example:-

$S \Longrightarrow_{rm} aABe \Longrightarrow_{rm} aAde \Longrightarrow_{rm} aAbcde \Longrightarrow_{rm} abbcde$

Shift – Reduce Parser with Handle

Example

E → E + T | T

T → T * F | F

F → (E) | id

String : - id + id * id

Right – Most Sentential Form	Reduction Production	Handle
id + id * id	F → id	id
F + id * id	T → F	F
T + id * id	E → T	T
E + id * id	F → id	id
E + F * id	T → F	F
E + T * id	F → id	id
E + T * F	T → T * F	T * F
E + T	E → E + T	E + T
E		

Stack Implementation of Shift – Reduce Parser

Like a table-driven predictive parser, a bottom-up parser makes use of a stack to keep track of the position in the parse and a parsing table to determine what to do next.

Algorithm

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

Parser Actions

- **Shift**: - The next input symbol is shifted onto the top of the stack.
- **Reduce**: - Replace the handle on the top of the stack by the non – terminal.
- **Accept** :- Successful completion of parsing
- **Error** :- Parser finds a syntax error, and calls an error recovery routine

Example

E → E + T | T

T → T * F | F

F → (E) | id

String : - id + id * id

Stack	**Input**	**Action**
$	id + id * id $	Shift id
$id	+ id * id $	Reduce by F → id
$F	+ id * id $	Reduce by T → F
$T	+ id * id $	Reduce by E → T
$E	+ id * id $	Shift +
$E +	id * id $	Shift id
$E + id	* id $	Reduce by F → id
$E + F	* id $	Reduce by T → F
$E + T	* id $	Shift * (OR Reduce by E → T) CONFLICT
$E + T *	id $	Shift id

$E + T * id	$	Reduce by F → id
$E + T * F	$	Reduce by T → T * F
$E + T	$	Reduce by E → E + T
$E	$	Accept

Conflicts in Shift – Reduce Parsing

Some grammars cannot be parsed using shift-reduce parsing and result in conflicts. There are two kinds of shift-reduce conflicts:

Shift / Reduce Conflict

Here, the parser is not able to decide whether to shift or to reduce. Example: if A → ab → abcd, and the stack contains $ab, and the input buffer contains cd$, the parser cannot decide whether to reduce $ab to $A or to shift two more symbols before reducing.

Reduce / Reduce Conflict

Here, the parser cannot decide which sentential form to use for reduction. For example if A → bc and B → abc and the stack contains $abc, the parser cannot decide whether to reduce it to $aA or to $B

LR Parser

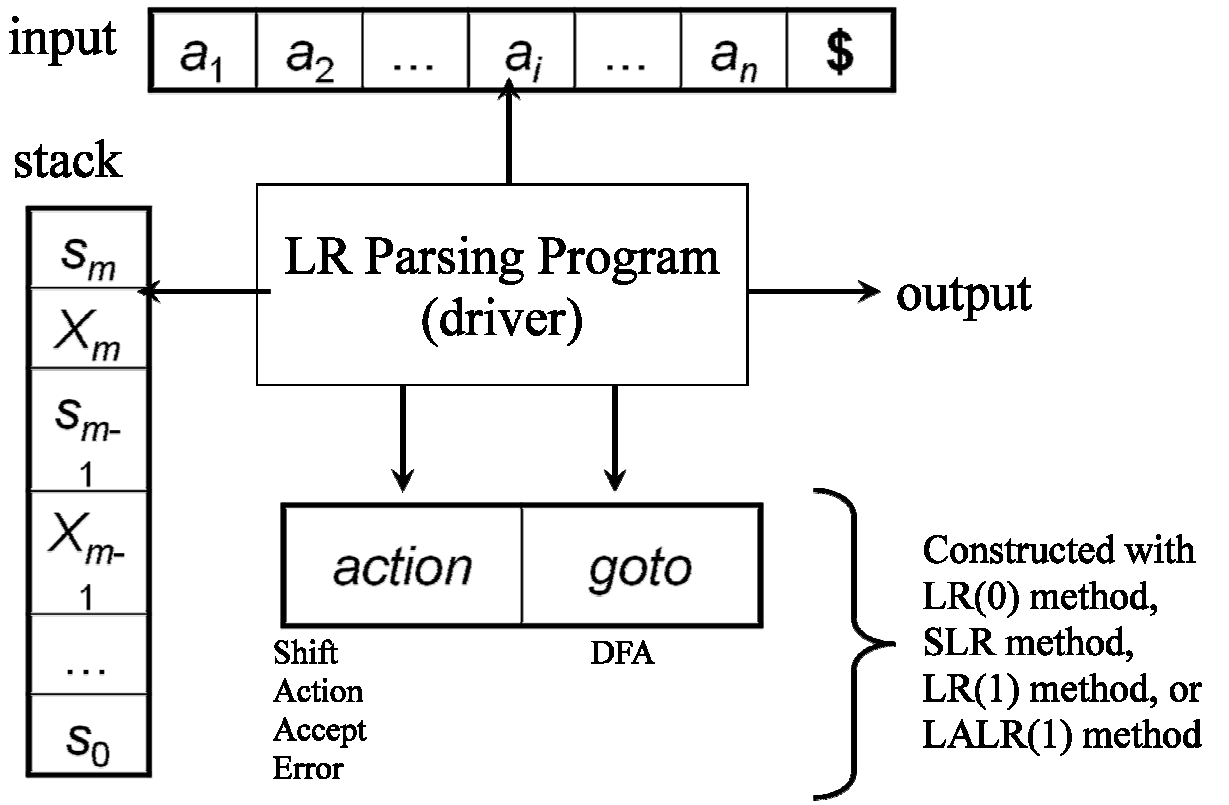
The class of grammars called LR(k) grammars have the most efficient bottom-up parsers and can be implemented for almost any programming language. The first L stands for left-to-right scan of the input buffer, the second R stands for a right-most derivation (left-most reduction), and k stands for the maximum of lookahead. If k is omitted, it 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 LL parsers. In fact, virtually all programming language constructs for which CFGs can be written can be parsed with LR techniques. The primary disadvantage is the amount of work it takes to build the tables by hand, which makes it infeasible to hand-code an LR parser for most grammars. Fortunately, there are LR parser generators like yacc (bison) that create the parser from an unambiguous CFG specification. The parser tool does all the tedious and complex work to build the necessary tables and can report any ambiguities or language constructs that interfere with the ability to parse it using LR techniques.

LR – parsers coves wide range of grammars.

- SLR (Simple LR parser)
- LR (Most general LR parser)
- LALR (Intermediate LR parser, Look ahead LR parser)

Note :- All LR parser use the same algorithm, the only difference is that their parsing table.

Structure of a General 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 $s_0X_1s_1X_2$: : :X_ms_m, where every s_i is called a "state" and every X_i is a grammar symbol (terminal or non-terminal).

If top of 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 →
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 of a DFA that recognizes the viable prefixes of the grammar.

LR(k) grammars are less stringent than LL(k) grammars. LR(k) grammars have to match the right side of a production by looking at its possible derivations wo maximum k levels. LL(k) grammars have to identify the right side of a production just by reading k terminals of the input string. LR(k) grammars describe a larger class of grammars.

LR Parser: Configurations / Actions

The configuration of a LR parser is a tuple comprising of the stack contents and the contents of the unconsumed input buffer. It is of the form ($s_0X_1s_1$ ….. X_ms_m; a_ia_{i+1} …… a_n\$).

1. If action[s_m, a_i] = shift s, the parser executes a shift move, entering the configuration ($s_0X_1s_1$ ……..$X_ms_ma_is$; a_{i+1} …….. a_n\$) shifting both a_i and the new state s.

2. If action[s_m, a_i] = reduce A → β, the parser executes a reduce move, entering the configuration $s_0X_1s_1$ …… $X_{m-r}s_{m-r}As$, a_ia_{i+1} ……. A_n\$). Here s = goto[$s_{m-r,}$ A] and r is the length of the handle β. (Note: If r is the length of β then the parser pops 2r symbols). After reduction, the parser outputs the production A → β.

3. If action[s_m, a_i] = accept then accept the grammar and stop.

4. If action[$s_{m,}$ a_i] = error then call error recovery routing.

Example

Grammar
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

⇒

state	**action**						**goto**		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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			

Now, let the input string be id * id + id $

Stack	Input	Action	Output
$0	id * id + id $	M[0, id] = s5, so shift 5	
$0 id 5	* id + id $	M[5, *] = r6, reduce 6 goto 3, here goto 3 is because, the length of |id| = r = 1, and popping 2r means popping id and 5, there is state 0 and goto[0, F] = 3	F → id
$0 F 3	* id + id $	Reduce 4 goto 2	T → F
$0 T 2	* id + id $	Shift 7	
$0 T 2 * 7	id + id $	Shift 5	
$0 T 2 * 7 id 5	+ id $	Reduce 6 goto 10	F → id
$0 T 2 * 7 F 10	+ id $	Reduce 3 goto 2,here r = |T*F| = 3 and pop 2r = 6, popping 6 element we got 0 and goto[0, T] = 2	T → T * F
$ 0 T 2	+ id $	Reduce 2 goto 1	E → T
$0 E 1	+ id $	Shift 6	
$0 E 1 + 6	id $	Shift 5	
$0 E 1 + 6 id 5	$	Reduce 6 goto 3	F → id
$0 E 1 + 6 F 3	$	Reduce 4 goto 9	T → F
$0 E 1 + 6 T 9	$	Reduce 1 goto 1	E → E + T
$0 E 1	$	Accept	

Constructing the Parsing Table : SLR Parsing Table

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

Item

An "item" (LR(0) item) is a production rule that contains a dot (•) somewhere in the right side of the production. For example, the production A → αAβ has four items:

A → • αAβ

A → α •Aβ

A → αA•β

A → αAβ•

The production A → ε, generates only one item A → •. An item represented by a pair of integers, the first giving the production and second the position of the dot. An item indicates how much of a production we have seen at a given point in the parsing process. For example, the first item above, indicates that w hope a string derivable from αAβ next on the input. The second item indicates that we have just seen on a input string derivable from α, and that we hope next to see a string derivable from Aβ. An item encapsulates what we have read until now and what we expect to read further from the input buffer. Sets of LR(0) items will be the states of action and goto table of the SLR parser. A collection of sets of LR(0) items (the canonical LR(0) collection) is the basis of constructing SLR parsers. To construct the canonical LR(0) collection we requires *augmented grammar* and two functions *closure* and *goto*.

Closure Operation

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

- Initially, every item in I is added to closure(I)
- If A →α •Bβ is in closure (I), and B → γ is a production, then add the item B → • γ to I if it is not already there. Apply this rule until no new items can be added to closure(I).

Example

Consider a grammar

E'→ E

E → E + T | T

T → T * F | F

F → (E) | id

If I = {[E' → • E]}, then closure(I) contains the items

E' → • E

E → •E + T

E → •T

T → •T * F

T → •F

F → • (E)

F → •id

GOTO Operation

In any item I, for all productions of the form A → α •X β that are in I, goto[I, X] is defined as the closure of all productions of the form A → αX• β

Example

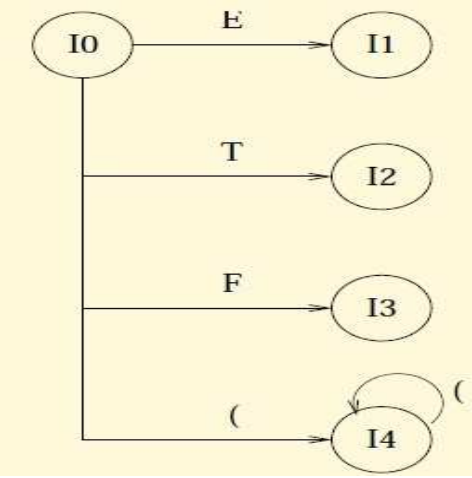
In above case, I_0 = closure({[E' → • E]}), then goto[I_0, E] = I_1

And I_1 = closure({[E' → E •],[E → E •+ T]})

SLR DFA

The goto operation defines a DFA over items of the SLR grammar.

Example



Algorithm for Constructing SLR Parsing Table

1. Given the grammar G, construct a an augmented grammar G' by introducing a production of the form S' → S, where S is the star symbol of G.
2. Let I_0 = *closure*({[S' → *S*]}). Starting from I_0, construct SLR DFA using *closure* and *goto*.
3. State *i* is constructed from I_i. The parsing actions for state *i* are defined as follows:
4. If *goto*[I_i, A] = I_j , where A is a non-terminal, then set *goto*[*i*, A] = *j*
    a. If *goto*[I_i, A] = $I_{j,}$, set *action*[*i*, a] = shift *j*, here *a* must be a teminal
    b. If I_i has the production of the form A → α •, then for all symbols a in FOLLOW(α), set *action*[*i*, a] = reduce A → α •. Here A should not be S'.
    c. If S' → *S* • is in I_i, then set *action*[*i*, $] = ***accept***
5. For all blank entries (i.e. *action* and *goto* entries not defined by steps 2 and 3), set entries to "***error***".
6. The start state s_0 corresponds to I_0 = *closure*({[S' → *S*]}).

Note : The set of all items {I_0, I_1,……,I_n} computed for the grammar is called the set of LR(0) items.

Example

E → E + T | T	Augmented Grammar is ⟹	E' → E
T → T * F | F		E → E + T | T
F → (E) | id		T → T * F | F
		F → (E) | id

Now,

I_0 = closure ([E' → E]) = E' → •E (here after dot(•), there is non – terminal so again expand it)

= E → •E + T

E → •T (again there is dot(•) after T so expand the production of T)

T → •T * F

T → •F (again same do for F)

F → • (E)

F → •id (not any non terminal remain after dot(•), so go for I_1)

Now for every non terminals and non terminals after dot(•), find *goto*

I_1 = goto(I_0, E) = E' → E•

E → E•+T

I_2 = goto(I_0, T) = E → T•

T → T• * F

I_3 = goto(I_0, F) = T → F •

I_4 = goto(I_0, ( ) = F → (•E)

E → •E + T

E → •T

T → •T * F

T → •F

F → • (E)

F → •id

I_5 = goto(I_0, id) = F → id•

Now, goto for I_0 is finished, now look for every members in I_1, that has dot(•) before it, and use goto, and continue so on.

I_6 = goto(I_1, +) = E → E+•T

T → •T * F

T → •F

F → • (E)

F → •id

I_7 = got(I_2, *) = T → T * •F

F → • (E)

F → •id

No possible goto for I_3 since there is not any member after dot(•)

I_8 = goto(I_4, E) = F → (E•)

E → E• + T

I_9 = goto(I_4, T) = E → T•

T → T• * F

Which is same as I_2, so no new state?

Similarly, goto(I_4, T) =I_2, goto(I_4, F) = I_3, goto(I_4, id) = I_5, goto(I_4, ( ) = I_4

Since no goto possible for I_5, go for I_6

I_9 = goto(I_6, T) = E → E+T•

T → T• * F

Since, goto(I_6, F) = I_3, goto(I_6, ( ) = I_4, goto(I_6, id) = I_5, go for others new state

I_{10} = goto(I_7, F) = T → T * F•

Since, goto(I_7, ( ) = I_4, goto(I_6, id) = I_5, go for others new state

I_{11} = goto(I_8,)) = F → (E) •

goto(I_8, +) = I_6, goto(I_9, *) = I_7

Now, finished stop and total numbers of state = 12

Rules:

1. If [A → α•aβ] is in I_i and *goto*[I_i, a] = I_j, then *action*[I_i, a] = s_j, remember ***a*** here is a terminal.
2. If[A → α•] is in I_i, then *action*[I_i, a] = reduce A → α for all ***a*** in FOLLOW(A), here A is not S'.
3. If [S' → S] is in I_i, then *action*[I_i, $] = *accept*

Now let us build the SLR parsing table

State	action						goto		
	id	+	*	(	)	$	E	T	F

Now, among all I_i, let us chose those productions which match either A → α•aβ , or A → α•, or S' → S. For I_0, the production matching in our rules are only F → • (E) and F → •id

The first, F → • (E), matches the production A → α•aβ, so apply rule 1. i.e. here *goto*[I_0, ( ] = I_4, that means j = 4, so shift with 4, here the terminal value of ***a*** is (.

Similar case for second production, shift with 5, since *goto*[I_0, id) = I_5,

Also, *goto*[I_0, E] = I_1, *goto*[I_0, T] = I_2, *goto*[I_0, F] = I_3,

Now the parsing table looks like

State	**action**						**goto**		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3

For I_1, two productions E' → E• and E → E•+T matches, where first followed Rule 3, since E is the starting symbol in our grammar, while next follow Rule 1.

State	**action**						**goto**		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			

For I_2, the matching productions are E → T•, T → T• * F. The second production works as above, and *goto*[I_2, *] = I_7, shift with 7. For first production, we have to apple Rule 2. As comparing with A → α•, we have to calculate FOLLOW(A) = FOLLOW(E) = {+,), $}. Now, *action*[$I_2$, +] = *action*[$I_2$,)] = *action*[$I_2$, $] = reduce by production E → T , which is production number 2 in our grammar, so table looks like now

State	**action**						**goto**		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2			r2	r2			

Now as continuing same process, we get the following final table

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2			r2	r2			
3		r4	r4		r4	r4			
4		r6	r6		r6	r6			
5	s4			s5			8	2	3
6	s4			s5				9	3
7	s4			s5					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

SLR and Ambiguity (Limitation of SLR)

Every SLR grammar is unambiguous. But there exist certain unambiguous grammars that are not SLR. In such grammar there exist at least one multiply defined entry *action[i,a],* which contains both a shift and reduce directive. The SLR technique still leaves something to be desired, because we are not using all the information that we have at our disposal. When we have a completed configuration (i.e., dot at the end) such as A→ α •, we know that this corresponds to a situation in which we have α as a handle on top of the stack which we then can reduce, i.e., replacing α by A. We allow such a reduction whenever the next symbol is in Follow(A). However, it may be that we should not reduce for every symbol in Follow(A), because the symbols below α on the stack preclude u being a handle for reduction in this case. In other words, SLR states only tell us about the sequence on top of the stack, not what is below it on the stack. We may need to divide an SLR state into separate states to differentiate the possible means by which that sequence has appeared on the stack. By carrying more information in the state, it will allow us to rule out these invalid reductions.

Consider a grammar

S → L = R

S → R

L → * R

L → id

R → L

For this grammar, we would get the following SLR parsing table

state	**action**				**goto**		
	id	=	*	$	S	L	R
0	s5		s4		1	2	3
1				accept			
2		s6/r5		r5			
3				r2			
4	s5		s4			8	7
5		r4		r4			
6			s4	s5		8	9
state	**action**				**goto**		
	id	=	*	$	S	L	R
7		r3		r3			
8		r5		r5			
9				r1			

In state 2, *action[2, =]* = s6 and *action[2, =]* = r5, it is seen that there is shift reduce conflict. Suppose the input string was of the form id = ….., where *id* was reduced to L and "=" is the next input symbol, and the state of the parser is in I_2. Because R → L• is contained in I_2, the parser tries to reduce L to R according to the reduction rule. The sentential form now becomes R = ….. But we see that no right hand side of a rule has R = ……, anywhere in the grammar. To remove such ambiguity we use LR(1) grammar.

LR(1) Grammar

In order to obviate invalid reductions, the general form of an item is of the form [A → α•β , a]. In this case, the second term ***a*** has no effect, but in an item of the form [A → α•, a], reduction is performed using 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" (which is of length 1).

LR(1) item = LR(0) item + lookahead

Computation of Closure(I) for 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]}).

Construction of Canonical LR(1) Parsing Table

1. Given the grammar G, construct a an augmented grammar G' by introducing a production of the form S' → S, where S is the star symbol of G.
2. Construct the set C = {I_0, I_1, ……., I_n} of LR(1) items.
3. If [A → α•aβ, b] ∈ I_i and *goto(I_i, a)* = I_j then set *action[i, a]* = $shift_j$
4. If [A → α•, b] ∈ I_i then set *action[i, b]* = reduce A → α (apply only if A ≠ S')
5. If [S' → S•, $] is in I_i then set *action[i, $]* = accept.
6. If *goto(I_i, A)* = I_j then set *goto[i, A]* = j
7. Repeat 3 – 6 until no more entries added.
8. The initial state I is the I_i holding item [S' → S•, $]

Example

Consider the grammar

S → CC

C → eC

C → d

Augment the grammar

S' → S

S → CC

C → eC

C → d

Find I_0 = Closure (S' → S, $)

S' → •S, $

Comparing it with A → α •Bβ, a, we get A = ε, α = ε, S = B, β = ε, a = $. Now, FIRST($\beta$a) = FIRST($) = $

Now apply the productions of B = S, with lookahead FIRST(βa) = $, That means

S → •CC, $

Now again, comparing it with A → α •Bβ, a, we get A = ε, α = ε, S = C, β = C, a = $. Now, FIRST($\beta$a) = FIRST(C$) = FIRST(C) = {e, d}. Now

C → eC, e

C → eC, d

C → d, e

C → d, d

We can rewrite this as C → eC, e / d and C → d, e / d

So, I_0 = S' → •S, $

S → •CC, $

C → •eC, e / d

C → •d, e / d

I_1 = goto(I_0, S) = S' → S•, $

I_2 = goto(I_0, C) = S → C•C, $

C → •eC, $

C → •d, $

I_3 = goto(I_0, e) = C → e•C, e / d

C → •eC, e / d

C → •d, e / d

I_4 = goto(I_0, d) = C → d•, e / d

I_5 = goto(I_2, C) = S → CC•, $

I_6 = goto(I_2, e) = C → e•C, $

C → •eC, $

C → •d, $

I_7 = goto(I_2, d) = C → d•, $

I_8 = goto(I_3, C) = C → eC•, e / d

goto(I_3, e) = I_3

goto(I_3, d) = I_4

I_9 = goto(I_6, C) = C → eC•, $

goto(I_6, e) = I_6

goto(I_6, d) = I_7

Now, let us build the canonical LR(1) parsing table.

For I_0, for production rule C → •eC, e / d, here ***a*** = e, and *goto[0, e]* = 3, From Rule 3, i.e. *action[0, e]* = shift***3***

Similarly for, C → •d, e / d, *action[0, d]* = shift***4***

For I_4, C → d•, e / d, here ***b = {e, d},*** so applying rule 4, *action[4, e] = action[4, d]* = reduce by production C → d.

Similarly, we get the following final table

state	**action**			**goto**	
	e	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		

LALR Parsing

Because a canonical LR(1) parser splits states based on differing lookahead sets, it can have many more states than the corresponding SLR parser. Potentially it could require splitting a state with just one item into a different state for each subset of the possible lookaheads. With LALR (*lookahead LR*) parsing, we attempt to reduce the number of states in an LR(1) parser by merging similar states. This reduces the number of states to the same as SLR, but still retains some of the power of the LR(1) lookaheads. LALR grammars are midway between SLR (simplest) and LR (most complex) grammar. They perform generalization over canonical LR item sets. A typical programming language generates thousands of states for canonical LR

parsers, while they generate only hundred of states for SLR and LALR parsers. LALR parser suffice for most programming languages requirements.

Union Operation on Items

Given an item of the form [A → α •Bβ, a], the first part (before comma) is called the "*core*" of the item. Given two states of the form I_i = {[A → α •, a]} and I_j = {[A → α •, b]}, the union of the states $I_{ij} = I_i \cup I_j$ = {[A → α •, a / b]}. I_{ij} will perform a reduce operation on seeing either a or b on the input buffer, whereas I_i and I_j are more stringent. 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 operations do not create any new shift-reduce conflicts, but can create new reduce-reduce conflicts. Shift operation depends only on the core and not on the next input symbol.

We may introduce a reduce / reduce conflict during the shrink process for the creation of the states of a LALR parser.

I_1 : A → α •, a B → β •, b I_2 : A → α •, b B → β •, c	⇒	I_{12} : A → α •, a A → α •, b B → β •, b B → β •, c	⇒	Reduce / reduce conflict in case of A → α •, band B → β •, b so it is not LALR

LALR Parsing Table Construction

1. Construct C = {I_0, I_1,……., I_n}, the collection of sets of LR(1) items.
2. For each core present in C, find all sets having the core and replace the sets by their union. Let C' = {J_0, J_1,……., J_m} be the resulting set of LR(1) items.
3. Construct the action elements of parsing table using the same method as for canonical LR(1) grammars.
4. If *J* is the union of k LR(1) items, J = $I_1 \cup I_2 \cup \ldots\ldots \cup I_k$, then the cores of *goto[I_1, X] goto[I_2, X]*, etc. are all same since I_1, I_2,……, I_k have the same core. Let *K* be the union of all sets of items having the same core as *goto[I_1, X]*. Then goto[J, X] = K.

Example

Consider the grammar

S → CC

C → eC

C → d

For this grammar, we already get all the items sets and among them I_3 and I_6, I_4 and I_7, I_8 and I_9 are equivalent. Because their core items are same

I_3	I_6	I_4	I_7	I_8	I_9
C → e•C, e / d C → •eC, e / d C → •d, e / d	C → e•C, $ C → •eC, $ C → •d, $	C → d•, e/d	C → d•, $	C → eC•, e/d	C → eC•,$

Now the equivalent LALR parsing table looks like

LR(1) Parsing Table					
state	**Action**			**goto**	
	e	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		

⇓

LALR Parsing Table					
state	**Action**			**goto**	
	e	D	$	S	C
0	s36	s47		1	2
1			accept		
2	s36	s47			5
3, 6	s36	s47			89
4, 7	r3	r3	r3		
5			r1		
8, 9	r2	r2	r2		

Let us see how *GOTO* is computed. Consider $GOTO(I_{36}, C)$. In the set of LR(1) items, $GOTO(I_3, C) = I_8$, and I_8 is now part of I_{89}, so we make $GOTO(I_{36}, C)$ be I_{89}. Similar cases for I_{36} and I_{47}. For another example, consider (I_2, e), an entry that leads to the shift action of I_2 on input ***e***. In the set of LR(1) items, $GOTO(I_2, e) = I_6$. Since I_6 is now part of I_{36}, $GOTO(I_2, e)$ becomes I_{36}. Thus, the entry for state 2 in above figure and input ***e*** is made s_{36}, meaning push state 36 onto the stack.

Efficient Construction of LALR Parsing Tables

Kernel and Non – Kernel Items

In order to devise a more efficient way of building LALR parsing tables, we define the terms kernel items and non-kernel items. Other than the initial item [S' → •S, $] no other item generated by a *goto* has a dot at the left end of the production. Such items (i.e. the initial item and all other items generated by *goto*) are called kernel items. Items that are generated by closure over kernel items have a dot at the beginning of the production. These items are called *nonkernel* items.

First, given an augmented grammar, compute the set of all LR(0) kernels.

Example

Consider the augmented grammar

S' → S

S → L = R

S → R

L → * R

L → id

R → L

The kernels of the set of LR(0) items are : I0 = { S' → •S }

I_1 = { S' → S• }

I_2 = { S → L• = R, R → L•}

I_3 = { S → R•}

I_4 = { L → *•R}

I_5 = { L → id•}

I_6 = { S → L= • R}

I_7 = { L → *R•}

I_8 = { R → L•}

I_9 = { S → L= R • }

Spontaneous and Propagated Lookaheads

Recall that in any itemset *I*, for any item of the form [A → α •Bβ, a], the closure was computed as the set of all items of the form [B → •γ, b] for all b in FIRST(βa). Let γ = Xη . Then the itemset *I* will have a *goto* defined on X that will take it to an itemset *I'* that contains an item of the form [B → X•η, b]. In the above *goto*, if ***b*** = ***a*** (i.e. if $\beta \Longrightarrow *\varepsilon$), then the lookahead symbol ***a*** is said to be "propagated" to *I'*. If ***b*** is not equal to ***a***, then the lookahead symbol (in *I'*) is said to be generated "spontaneously."

Algorithm

Given the kernel *K* of any itemset *I*, use the following algorithm for computing lookaheads:

1. For each item B → γ•δ in *K* do
2. *J'* = *closure*({[B → γ•δ, #]}) where # is a dummy lookahead symbol
3. If [A → α•Xβ, a] is in *J'* and ***a*** ≠ #, then lookahead ***a*** is generated spontaneously for item A → αX•β that is in *goto*[I, X]
4. If [A → α•Xβ, #] is in *J'* then lookaheads propagate from B → γ•δ in *I* to A → αX•β in *goto*[I, X].
5. Done

State	**Item**	**Lookaheads**			
		Initialization	Pass1	Pass2	Pass3
I_0	S' → •S	$	$	$	$
I_1	S' → S•		$	$	$
I_2	S → L• = R		$	$	$
I_2	R → L•		$	$	$
I_3	S → R•		$	$	$
I_4	L → *•R	=	= / $	= / $	= / $
I_5	L → id•	=	= / $	= / $	= / $
I_6	S → L= • R			$	$
I_7	L → *R•		=	= / $	= / $
I_8	R → L•		=	= / $	= / $
I_9	S → L= R •				$

The column labeled "*initialization*" shows the spontaneously generated *lookaheads* for each kernel item. These are only the two occurrences of = , and the spontaneous *lookahead* $ for the initial item S' → •S. On the first pass, the *lookahead* $ propagates from S' → S in I_0 to the six items listed in above figure. The *lookahead* = propagates from L → *•R in I_4 to items L → *R•. in I_7 and R → L•. in I_8. It also propagates to itself and to L → id• in I_5, but these *lookaheads* are already present. In the second and third passes, the only new *lookahead* propagated is $, discovered for the successors of I_2 and I_4 on ***pass 2*** and for the successor of I_6 on ***pass 3***. No new *lookaheads* are propagated on pass 4, so the final set of *lookaheads* is shown in the rightmost column in above figure.

Exercises

1. Construct the LL(1) parsing table for following grammar.

   S → [A] S | ε

   A → {B} A | ε

   B → ( ) B | ε

2. Consider the grammar

   S → (L) | a

   L → L, S | S

   a. What are the terminals, non terminals and start symbol.

   b. Find the parse tree for the following sentences.

      i. (a, a)

      ii. (a, (a ,a))

      iii. (a, (a, a),(a, a)))

   c. Construct the rightmost and leftmost derivation of above sentences.

   d. What languages does this grammar support?

   e. Eliminate the left recursion.

3. Consider the grammar

   S → aSbS | bSaS | ε

   Show that this grammar is ambiguous by constructing two different leftmost derivations for the sentence *abab*.

4. Consider the grammar

   bexpr → *bexpr* **or** *bterm* | *bterm*

   bterm → *bterm* **and** *bfactor* | *bfactor*

   bfactor → **not** *bfactor* | *(bexpr)* | **true** | **false**

   a. Construct the parse tree for **not (true or false)**

   b. Show that this grammar generates all Boolean expressions.

   c. Construct the predictive parser for the above grammar.

5. Consider the grammar

   R → R '|' R | RR | R*| (R) | a | b

   Note that the first vertical bar is the "OR" symbol, not a separate between alternatives.

   a. Show that this grammar generates all regular expressions over the symbol ***a*** and ***b***.

b. Show that this grammar is ambiguous.

c. Construct the SLR parsing table.

6. The following grammar is proposed for if – then – else statements is proposed to remedy the dangling – else ambiguity.

   stmt → **if** *expr* **then** *stmt* | *matched_stmt*

   matched_stmt → **if** *expr* **then** *matched_stmt* **else** *stmt* | **other**

   Show that this grammar is still ambiguous.

7. Write the grammar for the following languages.

   a. The set of all bit strings such that every 0's is immediately followed by at least one 1.

   b. Binary string with equal numbers of 0's and 1's.

   c. Binary string with unequal numbers of 0's and 1's.

   d. Binary string in which 001 does not appear as substring.

   e. Binary string of the form **xy** where $\mathbf{x \neq y}$

   f. Binary string of the form **xx**.

8. Show that no left recursive grammar can be LL(1).

9. Show that no LL(1) grammar can be ambiguous.

10. Consider the following grammar.

    E → E + T | T

    T → TF | F

    F → F * | a | b

    a. Construct the SLR parsing table for this grammar.

    b. Construct the LALR parsing table.

11. Consider the grammar

    S → AaAb | BbBa

    A → ε

    B → ε

    Show that this grammar is LL(1) but not SLR(1)

12. Show that no LR(1) grammar can be ambiguous.

13. Show that the following grammar is LALR(1) but not SLR(1).

    S → Aa | bAc | dc | bda

    A → d

14. Show that following grammar is LR(1) but not LALR(1)

    S → Aa | bAc | Bc | bBa

    A → d

    B → d

15. Construct the SLR parsing table for the following grammar.

    E → E **sub** R | E **sup** E | {E} | c

    R → E **sup** E | E

Bibliography

- Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. 2008
- Notes from CDCSIT, TU of Samujjwal Bhandari and Bishnu Gautam
- Manuals of Srinath Srinivasa, Indian Institute of Information Technology, Bangalore