Principles of Top-Down Parsing

A top down parser creates a parse tree starting with the root, i. e., it starts a derivation from the start symbol of the grammar.

1. It could potentially replace any of the nonterminals in the current sentential form. However while tracing out a derivation , the goal is to produce the input string.

2. Since the input tokens are scanned from left to right , it will be desirable to construct a derivation that also produces terminals in the left to right fashion in the sentential forms.

3. A leftmost derivation matches the requirement exactly. A property of such a derivation is that there are only terminal symbols preceding the leftmost nonterminal in any leftmost sentential form.

4. The basic step in a top down parser is to find a candidate production rule (the one that could potentially produce a match for the input symbol under examination) in order to move from a left most sentential form to its succeeding left sentential form.

5. A top down parser uses a production rule in the obvious way - replace the lhs of the rule by one of its rhs.

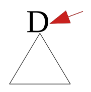
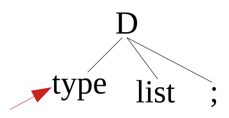
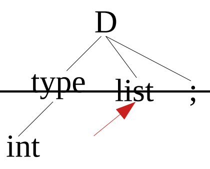
Example to Illustrate the Principles of Top-Down Parsing

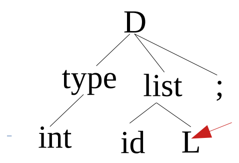
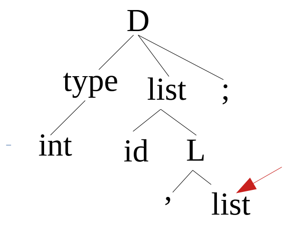
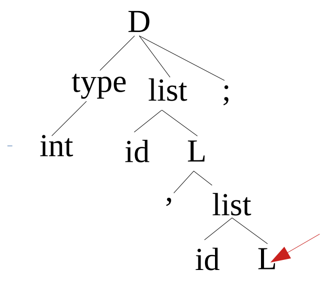
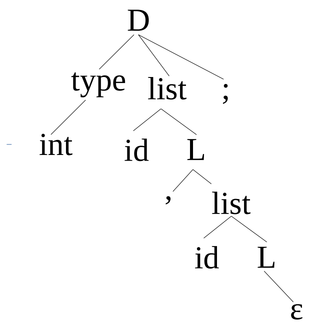
Top-down parsing of the sentence "**int a, b;**" for the C declaration grammar, named as G1, given below.

R1 : D → type list **;**	R2 **:** type → **int**
R3 **:** type → **float**	R4 **:** list → **id** L
R5 : L → **, id** L	R6 : L → ε

The start symbol of the CFG is the nonterminal D. The rules have been numbered for ease of reference.

The processing of a top down parser over the given CFG and the input can be understood by tracing the moves of the parser. The current token in the input is highlighted and an arrow points to the leftmost terminal in the sentential form.

Input	**Rule used with Explanation**	**Incremental Generation of Parse Tree**
int a , b ;	None	Initial Parse Tree 
int a , b ;	Leftmost nonterminal is D R1 : D → type list **;**	
int a , b ;	Leftmost nonterminal is type R2 **:** type → **int** int matches with the current token	

Input	**Rule used with Explanation**	**Incremental Generation of Parse Tree**
	get next token; mark leftmost nonterminal	
int a , b ;	Leftmost nonterminal is list R4 **:** list → **id** L a matches with current token id get next token; mark leftmost nonterminal	
int a , b ;	Leftmost nonterminal is L R5 : L → **,** list , matches with current token , get next token; mark leftmost nonterminal	
int a , b ;	Leftmost nonterminal is list R4 **:** list → **id** L b matches with current token id get next token; mark leftmost nonterminal	 D type list ; int id L , list id L
int a , b ;	Leftmost nonterminal is L R6 : L → ε There are no leftmost nonterminal left in the parse tree The nexttoken ; matches with the ‘;’ of rule R1	 D type list ; int id L , list id L ε

At this stage the entire input string is exhausted and has been successfully expanded from the start symbol D. Not only the parsing has been successful, the process has also traced out a leftmost derivation.

Let us recap the production rules used for parsing. The nonterminal that has been expanded in a sentential form has been highlighted and it clearly shows that a leftmost derivation has been traced out while parsing the input sentence.

D ⇒ type list **;** ⇒ **int** list **;** ⇒ **int id** L **;** ⇒ **int id , id** L **;** ⇒ **int id , id ;**

The yield of the parse tree matches exactly the input string.

Q. How would Top Down parsing proceed if we had instead used the following equivalent grammar, say G2, given below ?

R1 : D → type list **;**
R2 **:** type → **int**
R3 **:** type → **float**
R4 **:** list → list **, id**
R5 : list → **id**

Analysing the CFG to extract Useful Information

Processing the production rules of CFG generates several useful information about its nonterminals. Let us consider the grammar G1.

R1 : D → type list **;**
R2 **:** type → **int**
R3 **:** type → **float**
R4 **:** list → **id** L
R5 : L → **, id** list
R6 : L → ε

- A nonterminal including the start nonterminal can derive several sentential forms using the grammar rules. The nonterminals are {D, type, list, L}

- Consider the following derivations from D

D ⇒ type list **;** ⇒ **int** list ; D ⇒ type list ; ⇒ **float** list ;

While D can derive many more sentential forms, it can be seen that the first terminals that can appear in any sentential form (leftmost, rightmost or an arbitrary one) that are derivable from D are just two, namely {int, float}.

Similar information about the other non-terminals of G1 can also be obtained. This information about a nonterminal, say A, is denoted by FIRST(A), and is very useful in parsing.

For example, when the first token of a input , say “short”, is checked for syntactic validity with respect to G1, a top down parser will try to match “short” with FIRST(D) ={int, float} and since this match fails, parser will declare an error.

In general, it may appear at a first glance that for a given G, to construct FIRST(A), for all A in G, one would have to generate all the sentential forms possibly in G. Fortunately that is not the case and these sets can be easily constructed using an algorithm that processes the rules iteratively and gets the desired information within a few iterations.

Construction of FIRST() sets

The set of terminals that can ever appear in any sentential from derivable from a string α, $\alpha \in (N \sqrt{} T)^*$, is denoted by FIRST(α).

Formally, this is defined as FIRST(α) = $\{ a \in T^* \mid \alpha \Rightarrow^* a\,\beta \text{ for some } \beta\}$.
It follows that if $\alpha \Rightarrow^* \varepsilon$, then $\varepsilon \in$ FIRST(α)
Some obvious facts about FIRST information are the following :

1. For a terminal $a \in T$, FIRST(a) = {a}
2. For a nonterminal A, if $A \rightarrow \varepsilon$, then $\varepsilon \in$ FIRST(A)
3. For a rule $A \rightarrow X_1X_2 \ldots X_r$

$$FIRST(A) = FIRST(A) \cup_k (FIRST(X_k)); 1 \le k \le r; \varepsilon \in X_i; 1 \le i \le k-1;$$

The expression above says that the contributions from the FIRST sets of X_2, X_3, .., X_{k-1} also need to be computed if all their preceding nonterminals can derive ε. For instance, the terminals in FIRST(X_5) are to be added to FIRST(A), only if, $\varepsilon \in$ FIRST(X_1) and $\varepsilon \in$ FIRST(X_2) and $\varepsilon \in$ FIRST(X_3) and $\varepsilon \in$ FIRST(X_4).

An Algorithm that computes FIRST sets of nonterminals for a general CFG follows.

Algorithm for FIRST sets
Input : Grammar G = (N, T, P, S)
Output : FIRST(A), $A \in N$
Method :

```
begin algorithm
for $A \in N$ do FIRST(A) = $\Phi$ ;
    change := true;
    while change do
    {   change := false;
        for $p \in P$ such that p is $A \rightarrow \alpha$ do
        {  newFIRST(A) := FIRST(A) $\cup$ Y; where Y is FIRST(α) from definition given earlier;
           if newFIRST(A) $\neq$ FIRST(A) then
               {change := true ; FIRST(A) := newFIRST(A);}
        }
end algorithm
```

Illustration of FIRST() Set Computation :

R1 : D → type list **;**	R2 **:** type → **int**
R3 **:** type → **float**	R4 **:** list → **id** L
R5 : L → **,** list	R6 : L → ε

Nonterminal	**Initialization**	**Iteration 1**	**Iteration 2**	**Iteration 3**
D	Φ	Φ	{int float}	{int float}
type	Φ	{int float}	{int float}	{int float}
list	Φ	{id}	{id}	{id}
L	Φ	{ε ,}	{ε ,}	{ε ,}
change	-	True	True	False

The working of the algorithm can be explained as follows. The initialization step is obvious. In the first iteration, rule R1 is used for FIRST(D) = FIRST(type list ;) = FIRST(type) = Φ

FIRST(type) = FIRST(int) = {int} using R2 and using R3, we get FIRST(type) = FIRST(type) ∪ FIRST(float) = {int float}. Since FIRST(type) has changed from Φ to {int float}, change is set to True. Similarly in the second iteration, FIRST(D) changes due to which change is again set to True. All the FIRST() sets in iteration 3 remain the same as they were at the end of iteration 2, which causes change to be assigned False and the algorithm terminates with the FIRST() sets as given at the end of iteration 3.

Q. Can you construct a derivation that generates a sentential form justifying the presence of a terminal in the FIRST(A) for any A?

Q. What is the worst complexity of the algorithm in terms of the quantities present in the grammar?

Another Useful Information From the CFG

Another information that is found to be very useful in several parsing methods (but not all) is also based on sentential forms that are generated by the grammar rules.

- Consider the following derivations from D using grammar G1

R1 : D → type list **;**	R2 **:** type → int
R3 **:** type → float	R4 **:** list → id L
R5 : L → **,** list	R6 : L → ε

D ⇒ type list **;** ⇒ int list ;   D ⇒ type list ; ⇒ type id L ;

D ⇒ type list ; ⇒ float list ;

list ⇒ id L ⇒ id , list ⇒ id , id L ⇒ **...**

- The information that is sought now is about the terminals that appear immediately to the right of a nonterminal in any sentential from of G1. Such information is named as FOLLOW(A) for a nonterminal A and gives the terminal symbols that immediately follow A in some sentential form of G1.

- For instance, using R1 it can be seen that FOLLOW(list) contains ';' similarly from the derivation sequence, **"**D ⇒ type list ; ⇒ type id L ;" we find that FOLLOW(L) contains ';'. It is intended to construct FOLLOW(A) for all A in N.

Rules for construction of FOLLOW() sets are the following.

F1 : For the start symbol of the grammar, $ ∈ FOLLOW(S). The rationale for this special rule is that it is essential to know when the entire input has been examined. A special marker, $, is appended to be actual input. Since S is the start symbol, only the end of input marker, $, can follow S.

F2 : If A → α B β is a grammar rule, then FOLLOW(B) ∪= {FIRST(β) − ε)}. This operator " ∪=" is used in the same sense as "+=" of C, that is the existing terminals in FOLLOW(B), add all the terminals in FIRST(β) to it. Ignore ε even if ε belongs to FIRST(β).

F3 : If either A → α B is a rule or if A → α B β is a rule and β ⇒*ε (that is, ε ∈ FIRST(β), then FOLLOW(B) ∪= FOLLOW(A)

The rule says that under the given conditions, whatever follows A also follows B and this fact can be verified by using the give rule in a derivation.

- The rules for FOLLOW() can be applied to write an algorithm that constructs FOLLOW(A) for all A in N iteratively, similar to the algorithm for construction of FIRST sets.

G1 is reproduced below for ready reference.

R1 : D → type list **;**	R2 **:** type → int
R3 **:** type → float	R4 **:** list → id L
R5 : L → **,** list	R6 : L → ε

- F1 applied to R1 gives FOLLOW(D) = {$}.
  F2 applied to R1 gives FOLLOW(type) ∪= FIRST(list) = {id};
  FOLLOW(list) ∪= FIRST{;} = {;}
- None of F1, F2 or F3 apply to R2 and R3.
- F3 applied to R4 gives FOLLOW(L) ∪= FOLLOW(list) = {;}
- F3 applied to R5 gives FOLLOW(list) ∪= FOLLOW(L) = {;}
- Processing of the FOLLOW() sets iteratively shows that the information converges at the end of iteration 2.

A in N	**FIRST()**	**Initialization**	**Iteration 1**	**Iteration 2**
D	{int float}	{$}	{$}	{$}
type	{int float}	Φ	{id}	{id}
list	{id}	Φ	{;}	{;}
L	{ε ,}	Φ	{;}	{;}
change		NA	True	False

Q. Can you construct a derivation that generates a sentential form justifying the presence of a terminal in the FOLLOW(A) for any A.?

Q. Write an algorithm for construction of FOLLOW() for all nonterminal symbols of a CFG, on the same lines as given for FIRST() for all nonterminals.

Q. What is the worst complexity of your algorithm in terms of the quantities present in the grammar?

TOP-DOWN PARSER CONSTRUCTION

Deterministic Top-Down Parser

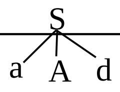
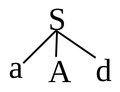
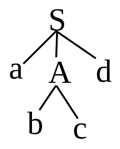
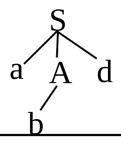
Consider the grammar given below with S as the start symbol.

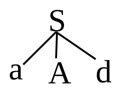
1.	$S \rightarrow aAd$
2, 3)	$A \rightarrow bc \mid b$

Parse the input **a c d** using a parser that is permitted to backtrack its moves, that is, in the case that it has made a wrong move and is not able to match the input, it can retract some move and try again. To make the backtracking deterministic, we shall try to backtrack the last move and start afresh. The objective is not to recommend backtracking parsers for implementation, but to reason why deterministic parsers, that never go back on its moves to parse an input, are so important for practical applications.

The starting point : Start nonterminal S and the input a c d

Parser chooses rule 1 as its move. The result is that S ⇒ a A d and i/p : a b d; highlighted symbols match, new configuration is : nonterminal A now has to match the remaining input: b d. The moves of a backtracking parser for this grammar and input are summarized in the table below.

Sentential Form	**Input**	**Rule and Derivation**	**Parse Tree**
S	a c d	Use the S ⇒ a A d	
⇒ a A d	a c d	token **a** matches; get next token and use the leftmost nonterminal	
⇒ a A d	a c d	Try one of the rule for A, A → bc	
⇒ a b c d	a c d	b does not match with c	Parse fails : backtrack the last rule used
⇒ a A d	a c d	Try the other rule for A, 3) A → b	

Sentential Form	**Input**	**Rule and Derivation**	**Parse Tree**
⇒ a b d	a c d	b does not match with c; Parse fails : backtrack the last rule; both rules for A fail; backtrack again.	
S ⇒ a A d	a c d	Try another rule for S but there is no other rule for S	S
S	a c d	No rule is left unexplored at this point.	Parse fails as there is no move left unexplored; hence “acd” does not belong to G

To compensate for no backtracking, some additional information is made available to a deterministic parser.

1. For instance, a token the parser is trying to match ; the current token being scanned in the input is known as the lookahead symbol.

2. Knowing the lookahead symbol helps the parser to make the selection. For example, let **a** be the lookahead symbol, and A be the leftmost nonterminal (lm nonterminal) in a leftmost sentential form. Let the rules for A be
$A \rightarrow a\alpha \mid b\beta$

It should be clear from the two rules for A, that the former rule, $A \rightarrow a\alpha$ is the right choice in this situation, since the other fails to match the lookahead token. In general, for a nonterminal A, the rule for A which contains the lookahead token as the first symbol in the rhs is a correct choice.

3. It is not necessary that for a nonterminal to have a terminal as the first symbol. How does a parser choose a rule in such a situation? Let the rhs of a rule, corresponding to a lm nonterminal, say A, have a nonterminal as the first symbol, such as $A \rightarrow C\gamma \mid B\beta$ which have nonterminals C and B respectively in the rhs of all the rules for A.

4. Given the lookahead symbol **t** to be matched, the rule that should be used to match **t** can not be determined directly from the rhs of the rules, $A \rightarrow C\gamma \mid B\beta$. If one could determine whether C (or B) derives a string whose first symbol is the lookahead, that is, whether **t** $\in$FIRST($C\gamma$) or **t** $\in$FIRST($B\beta$) then the choice is again possible, provided t belongs to one of two FIRST() sets. In the case that **t** $\in$ {FIRST($C\gamma$) $\cap$ **t** $\in$FIRST($B\beta$)}, then again there is a problem because both the rules can be applied for parsing and the grammar is then ambiguous for the top down parser.

Basic Steps of a top-down parser

1. Such a parser uses a rule by using its rhs a symbol at a time. A terminal symbol in the rhs must match the lookahead, else an error is reported.
2. A nonterminal symbol in the rhs calls for its expansion by choosing a suitable alternate from the rules associated with the lhs nonterminal.
3. Successful parse is indicated when the parser is able to consume all the tokens in the input. To ease the detection of end of input, a special symbol, $, is used. Otherwise an error is indicated at the point where the match failed to occur.

Left Recursive Grammar and Top-Down Parsing

Left recursive grammars could cause a top down parser to get into an infinite loop, even if the grammar is unambiguous and the input happens to be a valid sentence. The following example illustrates this fact.

Example : Consider the rule E $\rightarrow$ E + id | id and the input id + id + id \$. The grammar is left recursive because of the first rule.

- The parser could use the rule E $\rightarrow$ id, in which case the lookahead symbol being id, a match is found. The parser reports an error subsequently, since the next lookahead, +, does not match the rest of the rhs.

- If it uses the other rule, then the parser goes into an infinite loop. This is because if the parser move is to use the rule E $\rightarrow$ E + id , then after the expansion, the lm nonterminal is still E and the lookahead is id, so the same rule has to be applied again. Note that even if we had a backtracking top down parser, for this grammar the problem still remains.

- The conclusion is that left recursive are not to used with top down parsing.

We have already seen how left recursion can be removed from a CFG without changing the underlying language. Consider manual construction of a top down parser. It is assumed that the CFG is available and is free from left recursion.

- Since the rhs of the rules guide such a parser, a possible approach is to convert the rules directly into a program.
- For each nonterminal a separate procedure is written.
- The body of the procedure is essentially the rhs of the rules associated with the nonterminal. The basic steps of construction are as follows.

1. For each alternate of a rule, the rhs is converted into code symbol by symbol, as explained below.
    - If a symbol is a terminal, it is matched with the lookahead. The movement of the lookahead symbol on a match can be done by writing a separate procedure for it.
    - If the symbol is a nonterminal, call to the procedure corresponding to this nonterminal is made in its place.
2. Code for the different alternates of this nonterminal are appropriately combined to complete the body of the procedure.
3. The parser is activated by invoking the procedure corresponding to the start symbol of the grammar.

The process is illustrated through an example.

Example : Consider the rules given below :

exp $\rightarrow$ id exprime
exprime $\rightarrow$ + id exprime | ε

• For the first rule, exp $\rightarrow$ id exprime, we write a procedure named **exp** as follows. As we can observe, the rhs of exp is directly coded as the body of exp.

```
procedure exp()
begin
    if lookahead = id then
    begin
        match (id) ; exprime()      // same as rhs of exp written in code form
    end
    else error
    if lookahead = $ then report success // equivalent to $ ∈ FOLLOW(exp)
    else error
end ;
```

• The code for procedure match is now written. The match() procedure in the event of a match gets the next token, else it returns error because the match fails.

```
procedure match(token t);
begin
    if lookahead = t then
        lookahead := nexttoken ;
    else error
end ;
```

• The body of the final procedure for exprime() is written similarly.

```
procedure exprime ()
begin
    if lookahead = + then
        begin
        match (+) ;
        if lookahead = id then
            begin
            match ( id ) ; exprime ()
            end
        else error
        end
    else null ;
end ;
```

Recursive Descent Parser

Non-backtracking form of a top-down parser is also known as a predictive parser. The parser constructed manually above is a predictive parser.

A parser that uses a collection of recursive procedures for parsing its input is called a recursive descent parser. The procedures may be recursive (because of rules containing recursion). We constructed a recursive descent parser for the example grammar above.

Exercise : Modify the code for the procedures exp and exprime so that the rules that are used while parsing are also printed.

Comments on Recursive Descent Parsers

1. A recursive descent parser is easy to construct manually. However an essential requirement is that the language in which the parser is being written must support recursion.

2. Certain internal details of parsing are not directly accessible, for example
(a) the current leftmost sentential form that this parser is constructing,
(b) the stack containing the recursive calls active at any instant is not available for inspection or manipulation.

3. If there is a rule $A \to \alpha\beta1 \mid \alpha\beta2$, that is more than one alternates beginning with the same symbol in the rhs, then writing the code for procedure A is nontrivial. The method used in the example would fail to work for such cases.

4. A solution to the problem mentioned above is called **left factoring**. A rule such as

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

where γ does not begin with α, can be equivalently written as

$$A \to \alpha A_0 \mid \gamma$$
$$A_0 \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

5. A crucial question is whether one can always write a recursive descent parser (RD parser) for a context free grammar. The answer is no. The construction process, however, does not provide much help in characterizing the subclass of CFGs that admit a RD parser.

Table Driven Top Down Parser

We now study another top-down parsing algorithm. It is a non-recursive version of the recursive descent parser and happens to be the most popular among the top down parsers. It is commonly known as a LL(1) parser.

Writing an LL(1) Parser

Why the name LL(1) ?

The first L stands for direction of reading the input, L indicates left to right scanning of input; the next L indicates the derivation it uses (leftmost in this case) and 1 is the number of lookahead symbols used by the parser. LL(k), $k > 1$, that is LL parser with k lookahead symbols, is only of theoretical interest because of the increase in the size of parser and also its complexity.

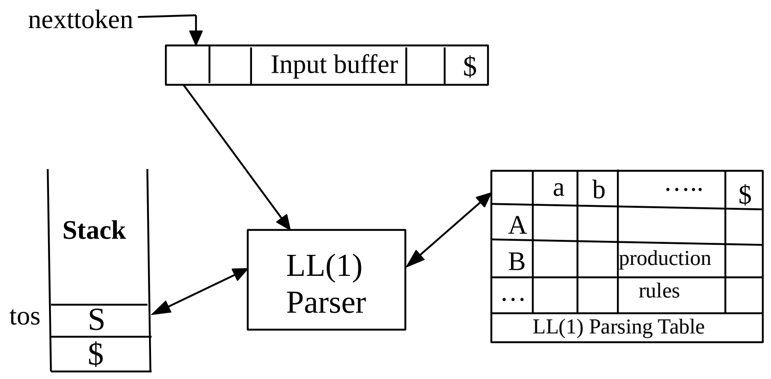
The components of an LL(1) parser is shown in Figure. The data structures employed by this parser are

- a stack
- a parsing table, and
- a lookahead symbol.

1. The purpose of the stack is to hold left sentential forms (or parts thereof). Since the requirement is to expand the leftmost nonterminal, the symbols in the rhs of a rule are pushed into the stack in the reverse order (from right to left).

2. The table has a row for every nonterminal and a column for every terminal (an entry for input-end-marker $ as well).

3. TABLE[A,t] either contains an error or a production rule.



Initial Configuration of LL(1) Parser

The working of LL(1) parser is given below, which is self explanatory.

tos	nexttoken	Parser actions
$	$	Successful parse; Halt
a	a	pop a; get nexttoken; continue
a	b	error
A	a	Table[A, a] = ‘-’ : where - denotes an error
A	a	Table[A, a] = A → XYZ pop A; push Z; push Y; push X; continue

Construction of Ll(1) Parsing Table

The parser consults the entry TABLE[A,t] when A is the lm nonterminal and t is the lookahead token. The table encodes all the critical parsing decisions and guides the parser. Such parsers are also known as table driven parsers.

The parsing algorithm is straightforward. The starting configuration is as shown in Figure.

- The top of stack element (tos) along with the lookahead token define a configuration of an LL(1) parser.
- The parser moves from one configuration to another by performing the actions given in the figure.
- The input is successfully parsed if the parser reaches the halting configuration.
- When the stack is empty and nexttoken is $, it corresponds to successful parse. To simplify detection of empty stack, $ is pushed at the bottom of the stack.
- Thus tos = nexttoken = $ is the condition for testing the halting configuration.
- The LL(1) Parser is a driver routine which refers to the parsing table, the lookahead token and manipulates the stack.

Illustration of LL(1) Table Construction for Example Grammar

p1 : decls → decl decls	p2 : decls → ε
p3 : decl → var list : type ;	p4 : list → id rlist
p5 : rlist → , id rlist	p6 : rlist → ε
p7 : type → integer	p8 : type → real

Construction of FIRST() sets

	Init	Iter 1	Iter 2	Iter 3
decls	Φ	{ε}	{ε var}	{ε var}
decl	Φ	{var}	{var}	{var}
list	Φ	{id}	{id}	{id}
rlist	Φ	{, ε}	{, ε}	{, ε}
type	Φ	{integer real}	{integer real}	{integer real}

Construction of FOLLOW() sets

	Init	Iter 1	Iter 2
decls	Φ	{$}	{$}
decl	Φ	{var}	{var}
list	Φ	{:}	{:}
rlist	Φ	{:}	{:}
type	Φ	{;}	{;}

To place the rule, “p1 : decls → decl decls”, FIRST (decls) = FIRST(decl decls) = {var}. This results in, TABLE[decls, var] = p1. Also because of “p2 : decls → ε”, using FOLLOW(decls) = {$}, we place TABLE[decls, $] = p2. The remaining entries of first column is ‘-’ because var ∉ {FIRST(list) ∩ FIRST(rlist) ∩ FIRST(type)}. Similarly since FIRST(list) = FIRST(id rlist) = {id}, we have the entry TABLE[list, id] = p4. The entries of the LL(1) table can be obtained by using the FIRST() and FOLLOW() sets for the rules. The final parsing table is given below.

	var	**id**	**:**	**;**	**,**	**real**	**integer**	**$**
decls	p1	-	-	-	-	-	-	p2
decl	p3	-	-	-	-	-	-	-
list	-	p4	-	-	-	-	-	-
rlist	-	-	p6	-	p5	-	-	-
type	-	-	-	-	-	p8	p7	-

Construction of LL(1) Parsing Table

All that remains is to use the FIRST() and FOLLOW() sets and construct the table row by roe.
of the table.

1. Create a row of the table for each nonterminal of the grammar.

2. The entry for the rule $A \rightarrow \alpha$ is done as follows.
   For each $a \in$ FIRST(α), create an entry TABLE[A, a] = { $A \rightarrow \alpha$}

3. If $\varepsilon \in$ FIRST(α), then create an entry TABLE[A, t] = {$A \rightarrow \alpha$}, where $t \in T \cup \{\$\}$ and $t \in$ FOLLOW(A).

4. All the remaining entries of the table are marked as “error”.

Remarks on LL(1) Parser

1. This method is capable of providing more details of the internals of the parsing process. For instance, the leftmost sentential form corresponding to any parser configuration can be easily obtained.

2. The syntax error situations are exhaustively and explicitly recorded in the table, LL(1) parser is guaranteed to catch all the errors.

3. How does one know whether a cfg G admits an LL(1) parser ? If the parsing table has unique entries, the resulting parser would work correctly for all sentences of L(G). However, if any entry in the table has multiple rules, the parser would not work and such a grammar is said to be LL(1) ambiguous.

4. The table construction leads to a characterization of the grammar that such a parser can handle. A grammar whose parsing table has no multiply defined entries is called a LL(1) grammar.

Characterization of LL(1) Parser

A characterization of an LL(1) grammar is a fallout of the theory of LL(1) parsing.

Let $A \rightarrow \alpha \mid \beta$ be two distinct production rules in a grammar. Purpose is to find out the conditions under which multiple entries for TABLE [A, a], for some a , occurs.

i) FIRST(α) $\cap$ FIRST(β) = {a}

ii) $\varepsilon \in$ { FIRST(α) $\cap$ FIRST(β)}, multiple entries for all $a \in$ FOLLOW(A)

iii) FOLLOW(A) $\cap$ FIRST(β) = {a} and $\varepsilon \in$ FIRST(α); then the corresponding table entry
has both the rules for A (mutentryz0; a similar situation exists for $\varepsilon \in$ FIRST(β).

iv) there are no other possibilities.

By complementing the conditions (i) to (iii) given above, we get a necessary and sufficient condition for a grammar to be LL(1).

An ambiguous grammar, such as the if-then-else grammar, is also LL(1) ambiguous (see figure below), but the converse is not necessarily true.

LL(1) Table for Dangling Else Grammar

Grammar Rules	p1 : S → i E t S S’	p2 : S →o
	p3 : S’ → e	p4 : S’ → ε
	p5 : E → c	

We have used abbreviations for the terminals : i for **if**; t for **then**; e for else for sake of convenience; E denotes an expression, however since it not essential for conditional statements, E is grounded to a terminal c; similarly statements of other language features, such as declarations, expressions, loops, function call, etc. are swept under the carpet and treated as a terminal named o.

Rule p1 sets the entry TABLE[S, i] to p1. Similarly for rule p2, we get TABLE[S, o] to p2. Rul3 p3 causes TABLE[S’, e] to p3. For rule p4, since S’ → ε, only for this rule we need FOLLOW(S’). FOLLOW(S’) = FOLLOW(S) = {e $}, because of which TABLE[S’, e] to p4 and also TABLE[S’, $] to p4. At this point, TABLE[S’, e] = {p3, p4}. Finally rule p5 causes TABLE[E, c] to be set to p5. The table is now complete and it is shown below.

	o	**c**	**e**	**i**	**t**	**$**
S	p2	–		p1		
S’			p3 p4			p4
E		p5				

The given if-then-else grammar is said to be LL(1) ambiguous because when S’ is the leftmost nonterminal and the lookahead token is **e**, this table driven parser has 2 valid moves, which is not permitted in a deterministic parser.

LL(1) Parser Generation
The manual construction of LL(1) parser reveals that the entire process can be automated. Figure below shows the internal steps for generating LL(1) parser automatically from the CFG description of a PL.

The first step in the construction process is to transform the given grammar G to an equivalent grammar G’ in case the given grammar has left recursion and /or left factoring. Note that the language of both grammars remain the same, that is, L(G) = L(G’).

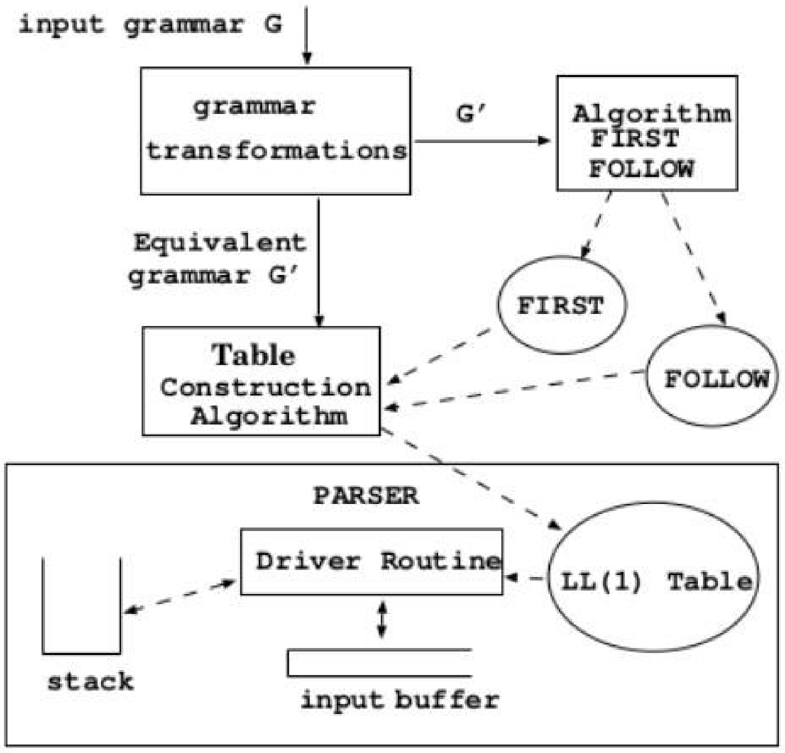
The transformed grammar G’ is now processed to construct the FIRST() and FOLLOW() sets for every nonterminals as explained earlier.

The nonterminals of the grammar A, which are not nullable, that is A does not derive ε, are processed along with the FIRST() sets to create some entries of the parsing table.

Finally, for the nullable nonterminals, A ⇒* ε, we use FOLLOW(A) to fill the remaining entries of the table. All the remaining entries are marked as error. This completes the construction of the LL(1) parsing table. A generic driver routine, which reads the lookahead token, and takes action based on the symbol (non-terminal or terminal) at the top of stack (tos) is written once for all included.

The driver routine, the parsing stack, along with the parsing table, and an input buffer, constitutes the top down parser. However it must be remembered that the parser is deterministic provided the table has at most one entry in all the positions, else the parser is not deterministic and the grammar is LL(1) ambiguous.

The parser generation process is illustrated in the following figure.


input grammar G
grammar transformations
G′
Algorithm FIRST FOLLOW
Equivalent grammar G′
FIRST
FOLLOW
Table Construction Algorithm
PARSER
Driver Routine
LL(1) Table
stack
input buffer