SYNTAX ANALYSIS

TEACHING MATERIAL

SYNTAX ANALYSIS

MOTIVATION

After lexical analysis, *syntax analysis* (or *parsing*) is the next phase in compiler design. Syntax analyser (or parser) is the name of that part of a compiler which performs this task.

What does a parser do?

- group tokens appearing in the input and attempt to identify larger structures in the program. This amounts to performing a *syntax check* of the program.

- make explicit the hierarchical structure of the token stream. Associated is the issue of representation - how should the syntactic structure of a program be explicitly captured ? The information may be required by subsequent phases.

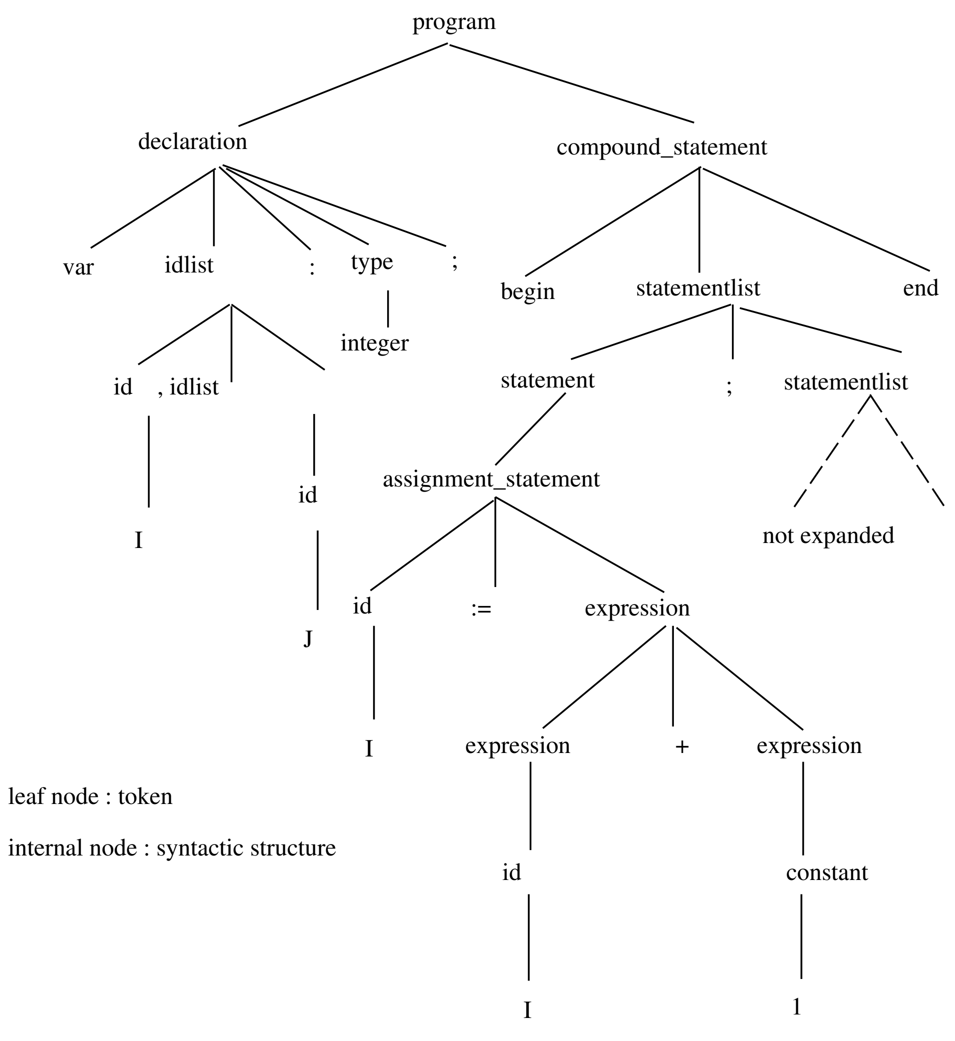
Let us try an intuitive parsing of the following program

```
var I,J: integer;
begin
     I := I + 1;
     J := J + 1
end.
```

SYNTAX ANALYSIS

MOTIVATION

FIGURE 3.1 : A possible grouping of tokens, exhibiting the hierarchy of syntactic structures, for the example program.



SYNTAX ANALYSIS

BASIC ISSUES IN PARSING

- **How to specify Programming Language (PL)Syntax ?**
  It is obvious that a parser must be provided with a description of PL syntax. How to describe the same is an important question and the related issues are :

  1. **the specification be precise and unambiguous**
  2. **the specification be complete, that is cover all the syntactic details of the language**
  3. **specification be such that it be a convenient vehicle for both the language designer and the implementer**

     We shall study one formalism called *context free grammar* **and examine the extent to which it meets these requirements.**

- **How to represent the input after it has been parsed ?**

  We shall study one representation known as *parse tree* **in this context.**

- **What are the parsing algorithms, how they work and what are their strengths/limitations ?**

  This is the primary concern of the module.

SYNTAX ANALYSIS

BASIC ISSUES IN PARSING

We shall discuss two different approaches to parsing , known as, *top-down* **and** *bottom- up*, **and study some parsing algorithms of both types.**

- **In syntax analysis phase, we are not interested in determining what the program does (***semantics* **) and hence such issues are not addressed here.**

How are parsers constructed ?

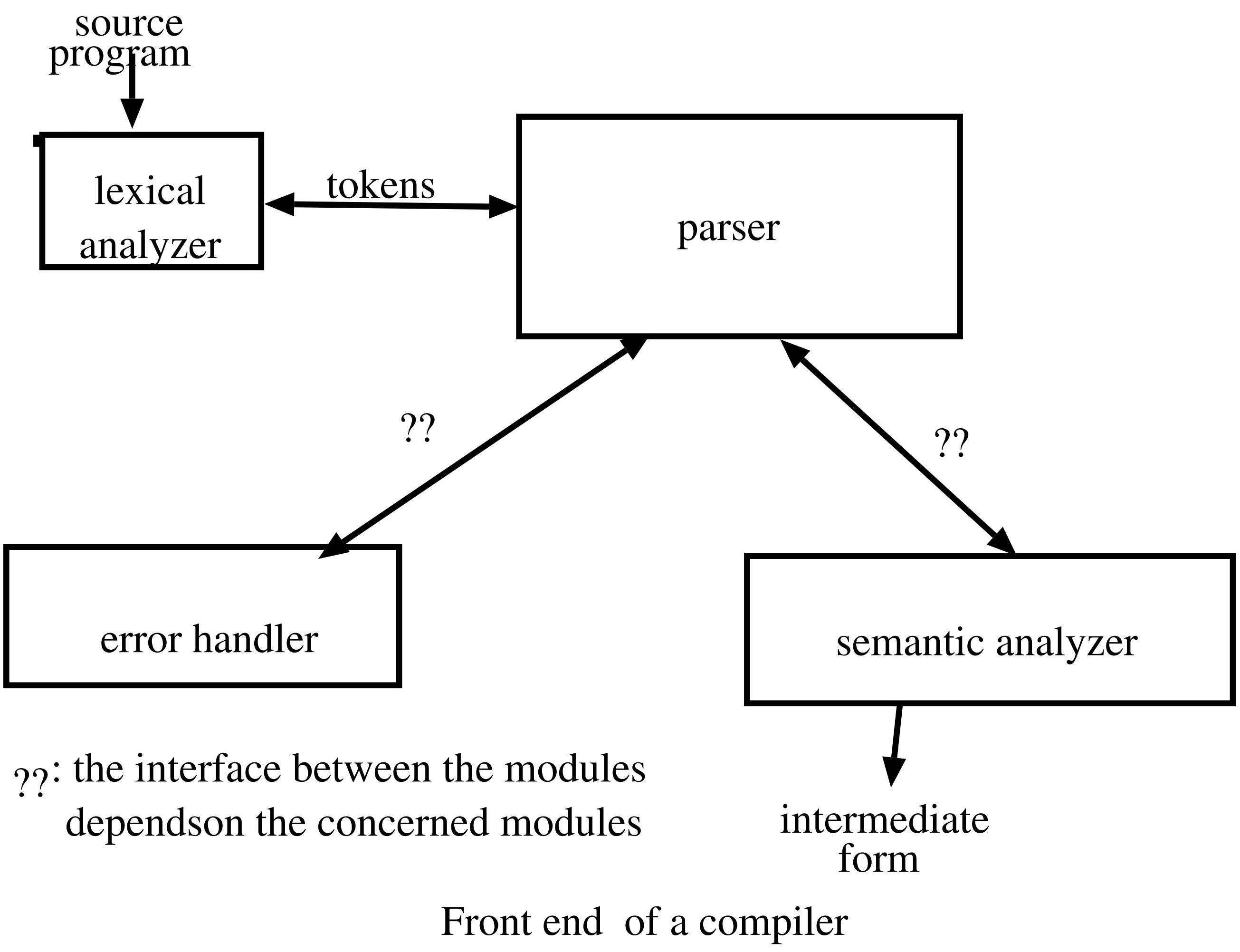
- **Till about 2 decades back, parsers (in fact all of the compiler) were written manually.**

- **A better understanding of the parsing algorithms has led to the development of special tools which can automatically generate a parser for a given PL.**
- **As we shall show later both top-down and bottom-up parsers can be automatically generated.**
- **A compiler generation tool, called YACC (Yet Another Compiler Compiler) generates a bottom-up parser and is available on Unix. Another such tool is BISON available in the GNU public domain software.**

SYNTAX ANALYSIS

BASIC ISSUES IN PARSING

FIGURE 3.2 : Interface of a parser with the rest of the compiler



SYNTAX ANALYSIS

SPECIFICATION OF PL SYNTAX

Program in any language is essentially a string of characters from its alphabet; however an arbitrary string is not necessarily a *valid* program.

- **The problem of specifying syntax is to determine those strings of characters that represent valid programs (programs that are syntactically correct). Rules which identify such valid programs spell out the syntax of the language.**

- **Recall that tokens are the lexical structures of a language and can be defined precisely and formally. This helps in the automatic generation of lexical analysers. What is the situation with syntactic structures and parsing ?**

- **Syntactic structures are grouping of tokens. The language definition provides the syntactic structures that are permitted. A few common ones are :**

expression	*declaration*
statement	*block*
compound statement	*procedure*
program	

SPECIFICATION OF PL SYNTAX

- Syntax of expressions is nontrivial - depends on the richness and the properties of the operators supported in the language.
- Let us informally describe a possible syntax for variable declarations in a Pascal-like language -
  - a stream of tokens that has keyword var as the first token
  - followed by one or more tokens of type identifier, separated by token comma
  - followed by token colon
  - followed by any one of the tokens, integer or real
  - followed by token semicolon

  Even for such a simple syntactic structure, the problems with informal description are evident. This is the motivation for formal specification of syntax.

To summarize, the following questions address some of the basic issues related to specification of syntax that a parser writer must be aware of.

SYNTAX ANALYSIS

SPECIFICATION OF PL SYNTAX

1. Do there exist formalisms that can be used to describe all the syntactic details of a PL ? If the answer is yes, then can the same formalism be used to write a parser for the language ?

2. If one uses a formalism that captures most of the syntactic details (but not all) of a PL and uses the same to write a parser, then when and how are the missing features taken care of ?

3. Given a formal mechanism , can the syntax of a language be specified uniquely ?

4. If the answer to above is no, then which among the several specifications should be preferred while constructing parsers and why ?

5. Is it useful for a compiler writer to know how to write syntactic specification that is suitable for parsing ?

BASIC CONCEPTS IN PARSING

CONTEXT FREE GRAMMAR

We introduce a notation called *Context Free Grammar* (CFG) informally and then give a formal definition.

- It is a notation for specifying programming language syntax.
- Identify the syntactic constructs of the language , such as expression, statement, etc. and express its syntax by means of *rewriting rules*

$$
\begin{array}{lcl}
\textit{declaration} & \rightarrow & \texttt{var}\ \textit{decl\_list} \\
\textit{decl\_list} & \rightarrow & \textit{list}\ :\ \textit{type}\ ; \\
\textit{list} & \rightarrow & \textbf{ID}\ \mid\ \textit{list}\ ,\ \textbf{ID} \\
\textit{type} & \rightarrow & \texttt{integer}\ \mid\ \texttt{real}
\end{array}
$$

Concatenation is the assumed operator between the symbols in the right hand side (rhs).

- Apart from tokens (also called as *terminals*), other symbols have been used in the rewrite rules above. These symbols called *nonterminals* (or *syntactic category*) do not exist in a source program. A nonterminal symbol occurs in the left hand side (lhs) of a rewrite rule.

BASIC CONCEPTS IN PARSING

CONTEXT FREE GRAMMAR

- **The rhs of a rewrite rule (also called a *production rule*) specifies the syntax of the lhs nonterminal. It may contain both kinds of symbols.**
- **Since the issue is to parse (and compile) programs, a nonterminal that specifies the syntax for a program must be present in the CFG.**

  program $\rightarrow$ **program ID (***list* **) ;** *decls compound_statement*

- **Other nonterminals are introduced to specify the various parts of the start symbol *program* in a structured manner.**
- **A CFG is formally defined in the following .**
  A CFG has four components, $G = (T, N, S, P)$
  - T **is a finite set of terminals (or tokens)**
  - N **is a finite set of nonterminals**
  - S **is a special nonterminal (from** N **) called the *start* symbol**
  - P **is a finite set of production rules of the form such as** $A \rightarrow \alpha$, **where** A **is from** N **and** α **from** $(N \cup T)^*$

BASIC CONCEPTS IN PARSING

CONTEXT FREE GRAMMAR

There can be more than one definition for a nonterminal (different rhs for the same lhs) in which case they are separated by $|$.

Why the term context free ?

1. *left_symbol* $\rightarrow$ *right_symbols* **is the only kind of rule permitted in a CFG; where** *left_symbol* **is a single nonterminal and** *right_symbols* **is a string from** $(N \cup T)^*$

2. **Rules are used to replace an occurrence of the lhs nonterminal by its rhs. In a CFG, the replacement is made regardless of the context of the lhs nonterminal (symbols surrounding it).**

Grammars are used to define languages. What is the relation between CFG of a PL and the PL itself ?

1. **Intuitively, a PL may be defined by the collection of all valid programs that can be written in that language.**

2. **A grammar is used to define a language in the sense that it provides a means of generating all its valid programs.**

BASIC CONCEPTS IN PARSING

CFG AND THE LANGUAGE DENOTED BY IT

3. Beginning with the start symbol of the grammar and using production rules repeatedly (for replacing nonterminal symbols), one can produce a string comprising terminals only. Such sequence of replacements is called a *derivation*.

4. The set of all possible terminal strings (elements of T^*) that can be derived from the start symbol of a CFG G is an important collection. Informally this set is the language denoted by G.

Example : consider $G = (T, N, S, P)$ with $N = \{list\}$, $S = list$, $T = \{ID\ ,\}$ and P containing a single rule

$$list \rightarrow list \text{ , ID } \mid \text{ ID}$$

A derivation is traced out as follows

$$\begin{array}{lll} list & derives & list \text{ , ID} \\ & derives & list \text{ , ID , ID} \\ & derives & \text{ID , ID , ID} \end{array}$$

This grammar generates terminal strings such as

ID ID, ID $\ldots$

The language denoted by G is one or more IDs separated by commas.

BASIC CONCEPTS IN PARSING

NOTATIONAL CONVENTIONS

Since it would be necessary to refer to terminals and nonterminals frequently, we adopt the following conventions.

Symbol type	Convention
single terminal	letters `a`, `b`, `c`, `operators` delimiters, keywords
single nonterminal	letters A, B, C and names such as *declaration* , *list* and S is the start symbol
single grammar symbol (symbol from $\{N \cup T\}$)	X, Y, Z
string of terminals	letters `x` , `y` , `z`
string of grammar symbols	α, β, γ
null string	ϵ

BASIC CONCEPTS IN PARSING

FORMAL DEFINITIONS

The concepts and terms that have been used informally are now formally defined. Let $A \rightarrow \gamma$ be a production rule.
Consider a string $\alpha\ A\ \beta$ from $(N \cup T)^*$.

1. Replacing the nonterminal A in the string above, yields the string $\alpha\ \gamma\ \beta$.

   Formally this is stated as $\alpha\ A\ \beta$ *derives* $\alpha\ \gamma\ \beta$ in a *derivation* of one step. A concise form of writing a one step derivation is $\alpha\ A\ \ \beta \Longrightarrow \alpha\ \gamma\ \beta$, where the symbol $\Longrightarrow$ stands for *derives in one step*.

2. If α_1, α_2, $\ldots$, α_n are arbitrary strings of grammar symbols, such that $\alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ \ldots \ \Longrightarrow \alpha_n$ then we say that α_1 derives α_n.

3. If a derivation comprises of zero or more steps , the symbol $\overset{*}{\Longrightarrow}$ is used. Clearly for any string α, $\alpha \overset{*}{\Longrightarrow} \alpha$ is true.

4. If a derivation comprises of one or more steps, the symbol $\overset{+}{\Longrightarrow}$ is used.

5. It is interesting to note that $\Longrightarrow$, $\overset{*}{\Longrightarrow}$ and $\overset{+}{\Longrightarrow}$ are *relations* over $(N \cup T)^*$ in the set theoretic sense. The last of these three relations is used to define the concept of a *language*.

FORMAL DEFINITIONS

6. The *language* $L(G)$ *denoted* by a context free grammar G is defined by $L(G) = \{ \text{ w } | \ S \overset{+}{\Longrightarrow} \text{ w, w} \in T^* \}$.
   Such a string w is called a *sentence* of $L(G)$.

7. A string α, $\alpha \in (N \cup T)^*$, such that $S \overset{*}{\Longrightarrow} \alpha$, is called a *sentential form* of G.

8. Grammars $G_1 = (T, N_1, S_1, P_1)$ and Grammars $G_2 = (T, N_2, S_2, P_2)$ are said to be *equivalent*, written as $G_1 \equiv G_2$, if they generate the same language, i.e., $L(G_1) = L(G_2)$.

EXAMPLE : Derivations and sentential forms are illustrated in Figure 3.3.

Consider the grammars given below. $G_1 = (T, N_1, L, P_1)$, where T = { , id }; N_1 = { L } and P_1 = { $L \rightarrow L$, id | id }.

$G_2 = (T, N_2, L, P_2)$, where N_2 = { L } and P_2 = {$L \rightarrow$ id , L | id}

$G_3 = (T, N_3, L, P_1)$, where N_3 = { L, L' } and P_3 = { $L \rightarrow$ id L' and $L' \rightarrow$, id L' | ϵ }
It can be seen that $G_1 \equiv G_2 \equiv G_3$.

BASIC CONCEPTS IN PARSING

DERIVATIONS AND THEIR USE

1. **A derivation of w from S is a proof that w $\in L(G)$.**

2. **Derivation provides a means for generating the sentences of $L(G)$.**

3. **For constructing a derivation from S, there are options at two levels**

   - **choice of a nonterminal to be replaced among several others**
   - **choice of a rule corresponding to the nonterminal selected.**

4. **Instead of choosing the nonterminal to be replaced, in an arbitrary fashion, it is possible to make an uniform choice at each step. Two natural selections are**

   - **replace the leftmost nonterminal in a sentential form**
   - **replace the rightmost nonterminal in a sentential form**

   The corresponding derivations are known as *leftmost* **and** *rightmost* **derivations respectively.**

5. **Given a sentence w of a grammar G, there are several distinct derivations for w.**

BASIC CONCEPTS IN PARSING

FIGURE 3.3 : An Example to illustrate the concepts

Grammar : $E \longrightarrow E + T \mid T$
$T \longrightarrow T * F \mid F$
$F \longrightarrow (E) \mid id$

Start symbol is E

Derivation of id in 3 steps $E \Longrightarrow T \Longrightarrow F \Longrightarrow id$

Also can be written as $E \overset{*}{\Longrightarrow} id$ or $E \overset{+}{\Longrightarrow} id$

A derivation of id + id * id

$$\underline{E} \Longrightarrow E+\underline{T} \Longrightarrow E+\underline{T}*F \Longrightarrow E+\underline{F}*F \Longrightarrow E+id*\underline{F} \Longrightarrow$$

sentential forms

$$\underline{E}+id*id \Longrightarrow \underline{T}+id*id \Longrightarrow \underline{F}+id*id \Longrightarrow id+id*id$$

(nonterminal chosen for expansion is underlined)

sentence

Leftmost derivation

$$\underline{E} \xRightarrow[\text{lm}]{} \underline{E}+T \xRightarrow[\text{lm}]{} \underline{T}+T \xRightarrow[\text{lm}]{} \underline{F}+T \xRightarrow[\text{lm}]{} id+\underline{T} \xRightarrow[\text{lm}]{}$$

$$id+\underline{T}*F \xRightarrow[\text{lm}]{} id+\underline{F}*F \xRightarrow[\text{lm}]{} id+id*\underline{F} \xRightarrow[\text{lm}]{} id+id*\underline{id}$$

Rightmost derivation

$$\underline{E} \xRightarrow[\text{rm}]{} E+\underline{T} \xRightarrow[\text{rm}]{} E+T*\underline{F} \xRightarrow[\text{rm}]{} E+\underline{T}*id \xRightarrow[\text{rm}]{} E+\underline{F}*id \xRightarrow[\text{rm}]{}$$

$$\underline{E}+id*id \xRightarrow[\text{rm}]{} \underline{T}+id*id \xRightarrow[\text{rm}]{} \underline{F}+id*id \xRightarrow[\text{rm}]{} id+id+id$$

BASIC CONCEPTS IN PARSING

DERIVATIONS AND PARSE TREES

There is an equivalent form of depicting a derivation that is pictorial in nature and is called a *parse tree*. A parse tree for a context free grammar, is a tree having the following properties

1. root of the tree is labeled with S
2. each leaf node is labeled by a token or by ϵ
3. an internal node of the tree is labeled by a nonterminal
4. if an internal node has A as its label and the children of this node from left to right are labeled with $X_1, X_2, \ldots, X_n$ then there must be a production

   $$A \rightarrow X_1 X_2 \ldots X_n$$

   where X_i is a grammar symbol.

5. the leaves of the tree read from left to right give the *yield* of the tree; essentially the sentence generated or derived from the root.
6. An example of a parse tree is given in Figure 3.4.

BASIC CONCEPTS IN PARSING

DERIVATIONS AND PARSE TREES

How to construct a parse tree from a derivation ?

Let $S = \alpha_1 \Longrightarrow \alpha 2 \Longrightarrow \ldots \Longrightarrow \alpha_n =$ **z be a derivation.**
The corresponding parse tree may be constructed by

1. **creating a root labeled with** $S = \alpha_1$
2. **for each sentential form,** α_i, $i \geq 2$, **constructing a parse tree with the yield of** α_i**. If the tree for** α_{i-1} **is available, the tree for** α_i **is easily constructed (by using induction), as given below.**
3. **Let** $\alpha_{i-1} = X_1X_2 \ldots X_r \Longrightarrow X_1X_2 \ldots X_{j-1}\beta X_{j+1} \ldots X_r = \alpha_i$
and $X_j \rightarrow Y_1Y_2 \ldots Y_s$ **be the rule used.**
   - **The node (leaf) corresponding to** X_j **in the parse tree is expanded by creating** s **children for this node, and**
   - **labeling these children with** $Y_1Y_2 \ldots Y_s$ **in the order from left to right.**

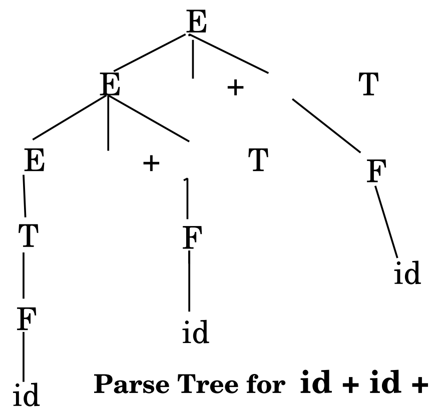
BASIC CONCEPTS IN PARSING

FIGURE 3.4 : Illustration of derivations and parse trees

Grammar : $E \longrightarrow E + T \mid T$

$T \longrightarrow T * F \mid F$

$F \longrightarrow (E) \mid id$



Parse Tree for id + id + id

leftmost derivation

$$\underline{E} \xRightarrow[lm]{} \underline{E}+T \xRightarrow[lm]{} \underline{E}+T+T \xRightarrow[lm]{} \underline{T}+T+T \xRightarrow[lm]{} \underline{F}+T+T \xRightarrow[lm]{} id+\underline{T}+T \xRightarrow[lm]{} id+\underline{F}+T \xRightarrow[lm]{} id+id+\underline{T} \xRightarrow[lm]{} id+id+\underline{F} \xRightarrow[lm]{} id+id+id$$

rightmost derivation

$$\underline{E} \xRightarrow[rm]{} E+\underline{T} \xRightarrow[rm]{} E+\underline{F} \xRightarrow[rm]{} \underline{E}+id \xRightarrow[rm]{} E+\underline{T}+id \xRightarrow[rm]{} E+\underline{F}+id \xRightarrow[rm]{} \underline{E}+id+id \xRightarrow[rm]{} \underline{T}+id+id \xRightarrow[rm]{} \underline{F}+id+id \xRightarrow[rm]{} id+id+id$$

BASIC CONCEPTS IN PARSING

DERIVATIONS AND PARSE TREES

The following summarize some interesting relations between the two concepts

1. Parse tree filters out the choice of replacements made in the sentential forms.
2. Given a derivation for a sentence, one can construct a parse tree for the sentence.
3. Even while several distinct derivations may exist for a given sentence, they usually correspond to a single parse tree.
4. Given a parse tree for a sentence, it is possible to construct a unique leftmost and a unique rightmost derivation.
5. Can a sentence have more than one distinct parse trees corresponding to it ?

   Construct examples, if such is possible.

AMBIGUOUS GRAMMAR

A context free grammar G that produces more than one parse tree for a sentence of $L(G)$ is defined to be an *ambiguous* grammar.

BASIC CONCEPTS IN PARSING

AMBIGUOUS GRAMMARS

Equivalently a context free grammar that has two or more leftmost (or rightmost) derivations for a sentence is an ambiguous grammar.

Ambiguous grammars are usually not suitable for parsing, because of the following reasons.

1. A parse tree would be used subsequently for semantic analysis; more than one parse tree would imply several interpretations and there is no obvious way of preferring one over another.
2. How does one detect that a given context free grammar is indeed ambiguous ?
   Since multiple parse trees, even for a single sentence, renders the grammar ambiguous, is an algorithmic solution feasible ?
3. What can be done with such grammars in the context of parsing?

   (i) Rewrite the grammar such that it becomes unambiguous.
   (ii) Use the grammar but supply disambiguating rules(cf. YACC).

BASIC CONCEPTS IN PARSING

WRITING A CFG

Given a PL, one may write several distinct but equivalent context free grammars for describing its syntax. There are several thumb rules that can be meaningfully used in writing a grammar that is better suited for parsing.

1. Many syntactic features of a language are expressed using recursive rules. Usually these can be written in either *left recursive* or *right recursive* form. While both forms may be equivalent in expressiveness, they have different implications in parsing. As an example, consider syntax for declarations :

   Using a left recursive rule (the first symbol in the rhs of a rule is the same nonterminal as that in the lhs)

   $D \rightarrow$ `var` *list* : *type* ;
   type $\rightarrow$ `integer` | `real`
   list $\rightarrow$ *list* , `id` | `id`

   Using a right recursive rule (the last symbol in the rhs of a rule is the same nonterminal as that in the lhs)

   $D \rightarrow$ `var` *list* : *type* ;
   type $\rightarrow$ `integer` | `real`
   list $\rightarrow$ `id` , *list* | `id`

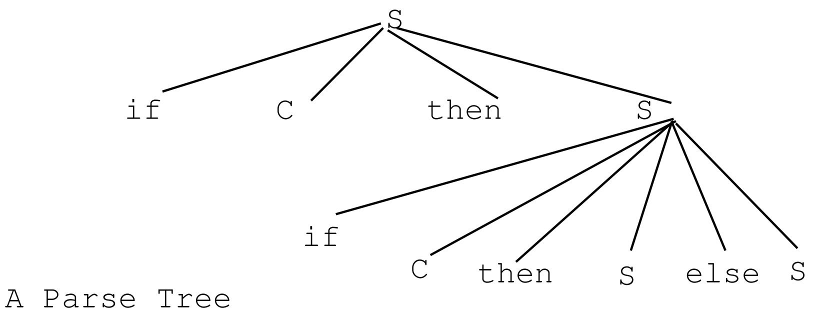
BASIC CONCEPTS IN PARSING

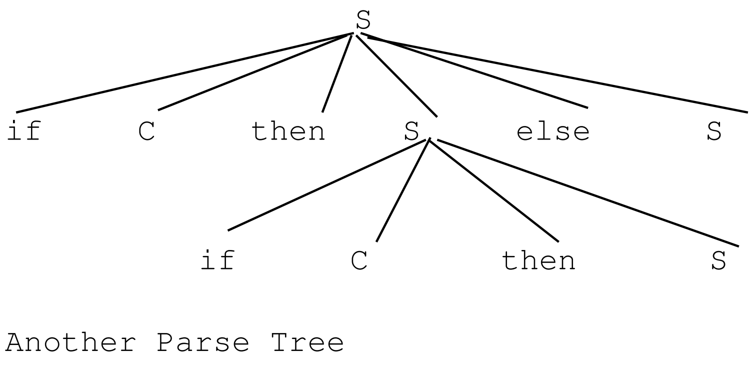
FIGURE 3.5 : AN AMBIGUOUS GRAMMAR

grammar is $\texttt{S} \longrightarrow \texttt{if C then S else S}$

$\texttt{S} \longrightarrow \texttt{if C then S}$

sentence is: `if C then if C then S else S`





BASIC CONCEPTS IN PARSING

WRITING A CFG

2. Writing a grammar for expressions; points of concern here are the operators and their properties.

   - *Associative property*: In the absence of parenthesis, how an expression such as a θ b θ c, is to be evaluated ? θ is some binary operation.

     An operator is classified as either *left* or *right* associative , depending upon whether an operand with same θ on both the sides is consumed by the θ placed to its left or right.
     Operators + , − , * and / are left associative while $\uparrow$ (or the assignment operator '=' in C) is right associative.
     For left(or right) associative operators, left (or right) recursive rules are useful.

   - *Precedence of operators* : Several operators have different precedences and an expression containing them has to be evaluated accordingly, else different interpretations arise.

     Language definition specifies the relative precedences of the permissible operators.

BASIC CONCEPTS IN PARSING

WRITING A CFG

- *Precedence of operators* (**continued**) **: Typically the operators { * , / } have higher precedence than { + , − }, rules for expression syntax using only these operators can be easily constructed as follows.**

  (i) A nonterminal for each of the two sets are chosen, say *term* **and** *exp* **respectively.**

  (ii) The rules for the operators with lower precedence are

  $$exp \rightarrow exp{+}term \mid exp - term \mid term$$

  Notice the left recursion and the use of 2 nonterminals.

  (iii) to write the rules for the other operator class, the basic units in expressions are needed. Another nonterminal, say, *factor***, is chosen for the purpose.**

  $$term \rightarrow term{*}factor \mid term/factor \mid factor$$

  (iv) The rules for the basic unit are :

  $$factor \rightarrow \texttt{id} \mid (exp)$$

BASIC CONCEPTS IN PARSING

REWRITING A CONTEXT FREE GRAMMAR

3. *Disambiguating a grammar* **: There are several useful transformations that make a cfg more amenable for parsing. Disambiguating a grammar is one of them.**

(i) The following grammar for expressions is ambiguous, but very concise. We know the thumb rules for disambiguating such grammars.

$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid E \uparrow \mid (E) \mid -E \mid \texttt{id}$

(ii) Consider another grammar given below.

$stmt \rightarrow \texttt{if}\ exp\ \texttt{then}\ stmt$
$\qquad\qquad \mid \texttt{if exp then}\ stmt\ \texttt{else}\ stmt \mid \texttt{other}$

(iii) To make the above grammar unambiguous, `else` has to be paired with the closest preceding unmatched `then`. How to incorporate such information into the grammar?

(iv) Let us examine the following input for the purpose.

```
i  t  i  t  i  t  e  i  t  e  e
```

`i,t,e` **stand for** `if`, `then`, `else` **respectively.**
How are the `else`**'s paired with the** `then`**'s?**

BASIC CONCEPTS IN PARSING

REWRITING A CONTEXT FREE GRAMMAR

(v) **Between any two consecutive `then` and `else`, only a complete `i  t  e` can occur, if any. This observation can be incorporated into the following grammar.**

$$
\begin{aligned}
&\mathit{stmt} \rightarrow \mathit{matched\_stmt} \mid \mathit{unmatched\_stmt} \\
&\mathit{matched\_stmt} \rightarrow \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{matched\_stmt}\ \texttt{else}\ \mathit{matched\_stmt} \\
&\qquad\qquad\qquad \mid \texttt{other} \\
&\mathit{unmatched\_stmt} \rightarrow \texttt{if exp then}\ \mathit{stmt} \\
&\qquad\qquad\qquad\quad \mid \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{matched\_stmt}\ \texttt{else}\ \mathit{unmatched\_stmt}
\end{aligned}
$$

4. *Left recursion removal* **: We have seen that PL grammars usually have recursive rules. We shall see later that** *left recursive grammars* **are unsuited for a class of parsing methods.**

   (i) **a context free grammar is** *left recursive* **if there exists a nonterminal A, such that**

   $$A \overset{+}{\Longrightarrow} A\alpha, \text{ for some } \alpha$$

   (ii) *Simple case* **: Consider the rule $A \rightarrow A\ \alpha \mid \beta$. An equivalent form without left recursion is**

   $$
   \begin{aligned}
   A &\rightarrow \beta\ A' \\
   A' &\rightarrow \alpha A' \mid \epsilon
   \end{aligned}
   $$

REWRITING A CONTEXT FREE GRAMMAR

(iii) if the recursion is direct (or *immediate*), then it is easy even for a general case like

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots A\alpha_m \mid \beta_1 \mid \beta_2 \ldots \mid \beta_n$

The transformed grammar in this is

$A \quad \rightarrow \beta_1\ A' \mid \beta_2\ A' \mid \ldots \beta_n\ A'$

$A' \quad \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_m A' \mid \epsilon$

(iv) The method listed above does not work for all left recursive grammars. Can you find the limitations of the suggested left recursion removal algorithm?

Exercises :

1. Write an algorithm based on (ii) and (iii) above for removing left recursion from an input cfg.
2. The recursion removal steps outlined above do not work for all recursive grammars. Can you find its limitations and construct examples in favour of the same.

INTRODUCTION TO PARSING

PARSING STRATEGIES

We begin with a formal definition of a parser.

A parser for a context free grammar G is a program P that when invoked with a string w as input, indicates that

1. either w is a sentence of G, i.e., w $\in L(G)$ (and may also give a parse tree for w)
2. or gives an error message stating that w is not a sentence (and may provide some information at or around the point of error).

Parsing Strategies

The two parsing strategies that we study are based on the following principles.

1. A parser scans an input token stream , from left to right, and groups tokens with the purpose of identifying a derivation, if one such exists.
2. In order to trace out such an unknown derivation, a parser makes use of the production rules of the grammar. Different parsing approaches differ in the choice of derivation and/or the manner they construct a derivation for the input.

INTRODUCTION TO PARSING

PARSING STRATEGIES

3. We have seen two standard derivations, viz., leftmost and rightmost, and also the way these can be represented through a parse tree. The two basic approaches to parsing can be easily explained using the concept of parse tree.

4. What are the possible ways in which a parse tree (or a tree data structure, in general) be constructed?

   - Create the parse tree from the root and expand it till all leaves are reached, i.e., in a top down manner. Parsers of this type are known as *top-down* parsers.
   - Create the parse tree from leaves upwards to the root, i.e., in a bottom up fashion. Parsers which use this strategy belong to the family of *bottom-up* parsers.

5. There being a direct relation between parse trees and derivations, both the parsing strategies can also be rephrased in terms of derivations.

In the subsequent slides of this module we now describe each strategy in detail in terms of

- Basic principles : the derivation on which it is based, how the derivation is constructed, relevant issues and problems for designing a deterministic parser of this family.

METHODS OF PARSING

PRINCIPLES OF TOP DOWN PARSING

- Manual construction of a parser : one or more parsing algorithms along with the data structures used, their limitations and strengths.
- Parser generators : how to automatically generate a parser from the cfg.

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.

METHODS OF PARSING

PRINCIPLES OF TOP DOWN PARSING

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 one.**

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

Example of Top-Down Parsing

Top-down parsing of the sentence `var a:integer;`
for the grammar given below

$$p_1 : D \longrightarrow \texttt{var}\ \textit{list} : \textit{type};$$
$$p_2 : \textit{type} \longrightarrow \texttt{integer}$$
$$p_3 : \textit{type} \longrightarrow \texttt{real}$$
$$p_4 : \textit{list} \longrightarrow \texttt{id}\ ,\ \textit{list}$$
$$p_5 : \textit{list} \longrightarrow \texttt{id}$$

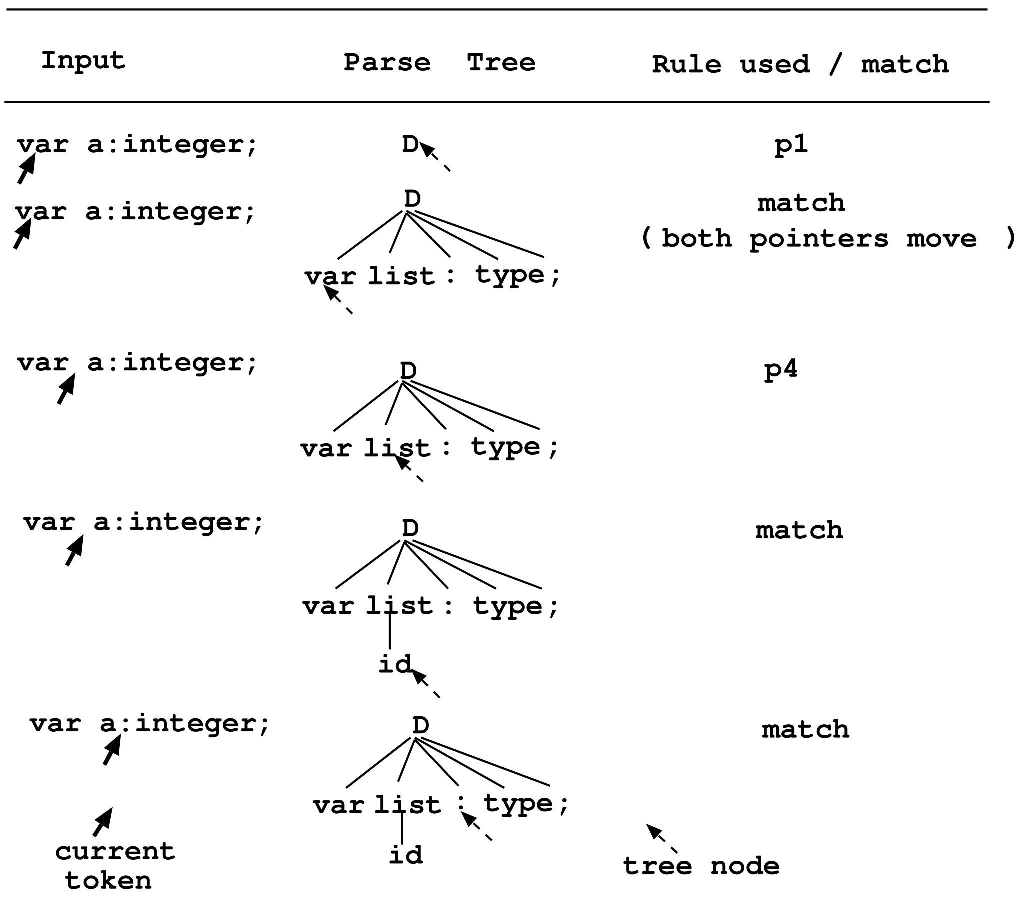
is illustrated in Figure 3.6.

PRINCIPLES OF TOP-DOWN PARSING

FIGURE 3.6 : Example of Top-Down Parsing

Declaration Grammar (see Slide 33)

Input : var a : integer;



TOP-DOWN PARSER CONSTRUCTION

BACKTRACKING AND NONBACKTRACKING PARSER

A top-down parser functions as given below.

1. The input sentence is parsed from left to right, examining each token in turn. A left-most derivation from the start symbol (initial leftmost sentential form) is initiated at the same time.

2. In order to expand the leftmost nonterminal in a sentential form, production rules are used. When there are several alternates for the leftmost nonterminal, a selection has to be made. There are two possible ways of effecting the selection.

   i) If backtracking is permitted, then the parser may try out all the alternates (in an arbitrary order) till it either finds a leftmost derivation (successful parse) or fails to do so (syntax error). An obvious question is the practical utility of a parser that uses backtracking.

   ii) A parser with no backtracking would be required to select the correct alternate, amongst the possible choices, in each step of the derivation process. Deterministic topdown parsers adopt this strategy. The crucial task is to identify the correct rule even before the rhs of the rule has been seen in the input.

TOP-DOWN PARSER CONSTRUCTION

DETERMINISTIC TOP-DOWN PARSER

Exercise : For the grammar given below

$S \rightarrow$ `a` A `d` $\qquad A \rightarrow$ `bc` $\mid$ `b`

Parse the input `a b d` using a backtracking parser.

Deterministic Top-Down Parser

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, A the lm nonterminal and the relevant rules be

$$A \rightarrow \texttt{a}\ \alpha \mid \texttt{b}\ \beta$$

- The former rule is the right choice in this situation. 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.

TOP-DOWN PARSER CONSTRUCTION

DETERMINISTIC TOP-DOWN PARSER

3. Let the rhs of a rule, corresponding to the lm nonterminal, have a nonterminal as the first symbol, such as

$$A \rightarrow C\gamma \mid B\beta$$

4. Which rule should be used to match a given lookahead can not be determined from the rhs of the rule. However, if one could determine if B (or C) derives a string whose first symbol is the lookahead then the choice is again possible.

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. A nonterminal symbol in the rhs calls for its expansion by choosing a suitable alternate from the rules associated with the lhs nonterminal
2. 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.

TOP-DOWN PARSER CONSTRUCTION

WRITING A TOP-DOWN PARSER

Left Recursive Grammar and Top-Down Parsing

Left recursive grammars could cause a top down parser to loop for ever, 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 $E \rightarrow E + \texttt{id} \mid \texttt{id}$
and the input $\texttt{id} + \texttt{id} + \texttt{id}\ \$$

i) **The grammar is left recursive because of the first rule.**

ii) **The parser could use the rule** $E \rightarrow \texttt{id}$**, 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.**

iii) **If it uses the other rule, then the parser goes into an infinite loop. Note that even if we had a backtracking top down parser, for this grammar the problem still remains.**

We consider manual construction of a top down parser. It is assumed that the cfg is available and is free from left recursion.

TOP-DOWN PARSER CONSTRUCTION

WRITING A TOP-DOWN PARSER

- 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.
   (a) 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.
   (b) 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.

TOP-DOWN PARSER CONSTRUCTION

WRITING A TOP-DOWN PARSER

3. **The parser is activated by invoking the procedure corresponding to the start symbol of the grammar.**
4. **The process is illustrated through an example.**

Example : Consider the rules given below :

$$exp \rightarrow \texttt{id}\ exprime$$
$$exprime \rightarrow + \texttt{id}\ exprime \mid \epsilon$$

- **For the first rule, we write a procedure named exp as follows.**

```
procedure exp;
    begin
        if lookahead = id then
           begin
                 match (id) ; exprime
           end
        else error
        if lookahead = $ then   report success
        else error
    end ;
```

TOP-DOWN PARSER CONSTRUCTION

PREDICTIVE PARSERS

- **The code for procedure match follows.**

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

- **The body of the other procedure 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 ;
```

TOP-DOWN PARSER CONSTRUCTION

RECURSIVE DESCENT PARSER

- Nonbacktracking form of a top-down parser is also known as a *predictive parser*. The parser constructed manually in the previous slides 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.

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

Comments on Recursive Descent Parsers

1. A recursive descent parser is easy to construct manually. However 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/manipulation

TOP-DOWN PARSER CONSTRUCTION

A TABLE DRIVEN TOP DOWN PARSER

3. If there is a rule $A \to \alpha\beta_1 \mid \alpha\beta_2$, that is more than one alternates have 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 $A \to \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma$
   can be equivalently written as
   $A \to \alpha A'|\gamma$
   $A' \to \beta_1|\beta_2|\ldots|\beta_n$

5. A crucial question is whether one can always write a recursive descent 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 such a 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.

TOP-DOWN PARSER CONSTRUCTION

WRITING AN LL(1) PARSER

Why the name LL(1) ?

The first L stands for left to right scanning of input; the next L indicates the derivation it uses (leftmost) and 1 is the the number of lookahead symbols used by the parser.

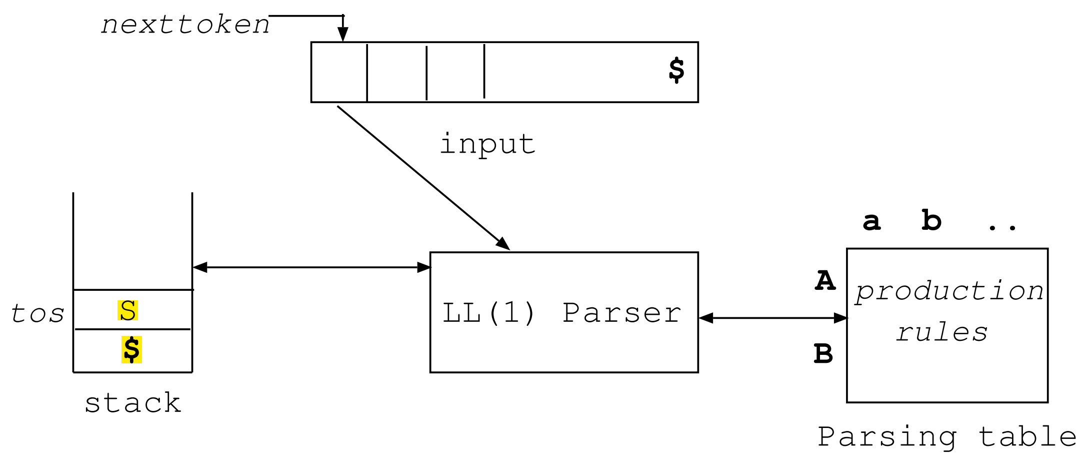
- The components of an LL(1) parser is shown in Figure 3.7. The data structures employed by the parser are -
  - a stack
  - a table
  - a lookahead symbol.

i) (a) the purpose of the stack is to hold left sentential forms (or parts thereof).
(b) since the requirement is to expand the leftmost non-terminal, the symbols in the rhs of a rule is pushed into the stack in the reverse order (from right to left).

ii) (a) The table has a row for every nonterminal and a column for every terminal(an entry for input endmarker $ as well).
(b) TABLE[A,t] either contains an error or a production rule.

TOP-DOWN PARSER CONSTRUCTION

FIGURE 3.7 : COMPONENTS OF AN LL(1) PARSER



Initial Configuration

Working of an LL(1) Parser

tos	nexttoken	parser action
$	$	successful parse; halt
a	a	pop a ; move nexttoken; continue
a	b	error
A	a	TABLE[A, a] = error
A	a	TABLE[A, a] = A $\longrightarrow$ XYZ pop A; push Z; push Y;push X; continue

TOP-DOWN PARSER CONSTRUCTION

CONSTRUCTION OF LL(1) PARSING TABLE

ii(c) 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.

iii) The parsing algorithm is straightforward. The starting configuration is as shown in Figure 3.7.
(a) The top of stack element (tos) alongwith the lookahead token define a *configuration* of an LL(1) parser.
(b) The parser moves from one configuration to another by performing the actions given in the figure.
(c) The input is successfully parsed if the parser reaches the halting configuration.

iv) 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

v) The LL(1) Parser is a driver routine which refers to the parsing table, the lookahead token and manipulates the stack.

FIGURE 3.8 : LL(1) Table for example grammar

grammar

p1, p2 : decls $\rightarrow$ decl decls | ε

p3 : decl $\rightarrow$ var list : type ;

p4 : list $\rightarrow$ id rlist

p5 , p6 : rlist $\rightarrow$, id rlist | ε

p7 , p8 : type $\rightarrow$ integer | real

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

e denotes an error entry

TOP-DOWN PARSER CONSTRUCTION

FIGURE 3.9 : Working of an LL(1) Parser

Parsing Table given in Figure 3.8

Input : var id,id real;

STACK	REMAINING INPUT	MOVE
$\underline{\texttt{decls}}$	$\underset{\texttt{---}}{\texttt{var}}$ id,id real;	use p1
decls $\underline{\texttt{decl}}$	$\underset{\texttt{---}}{\texttt{var}}$ id,id real;	use p3
decls;type:list $\underline{\texttt{var}}$	$\underset{\texttt{---}}{\texttt{var}}$ id,id real;	match
decls;type:$\underline{\texttt{list}}$	$\underset{\texttt{---}}{\texttt{id}}$,id real;	use p4
decls;type:rlist $\underline{\texttt{id}}$	$\underset{\texttt{---}}{\texttt{id}}$,id real;	match
decls;type $\underline{\texttt{rlist}}$	$\underset{\texttt{--}}{\texttt{,}}$id real;	use p5
decls;type:rlist id$\underline{\texttt{,}}$	$\underset{\texttt{--}}{\texttt{,}}$id real;	match
decls;type:rlist $\underline{\texttt{id}}$	$\underset{\texttt{---}}{\texttt{id}}$ real;	match
decls;type:$\underline{\texttt{rlist}}$	$\underset{\texttt{----}}{\texttt{real}}$;	error

Top of the stack element is shown as $\underline{\texttt{symbol}}$

nexttoken is shown as $\underset{\texttt{------}}{\texttt{token}}$;

Parser halts in an error configuration

TOP-DOWN PARSER CONSTRUCTION

CONSTRUCTION OF LL(1) PARSING TABLE

A cfg and its LL(1) parsing table is given in Figure 3.8. The working of the parser for the input of Figure 3.8 is illustrated in Figure 3.9.

- It should be clear at this point that the heart of an LL(1) parser is the table.
- If we know how to construct the table manually, then the rest of the parser is easily written.
- The key issue is to determine for every rule the terminal symbols for which it should be selected.
  - This collection of terminal symbols is called FIRST set and defined as follows

    FIRST(α) = $\{ \texttt{a} \mid \alpha \overset{*}{\Longrightarrow} \texttt{a}\ \beta$ for some $\beta\ \}$
    If $\alpha \overset{*}{\Longrightarrow} \epsilon$, then $\epsilon \in$ FIRST(α)

Some obvious facts about FIRST information are

1. For a terminal $\texttt{a}$, FIRST($\texttt{a}$) = $\{ \texttt{a} \}$
2. For a nonterminal A, $A \rightarrow \epsilon$, $\epsilon \in$ FIRST(A)

TOP-DOWN PARSER CONSTRUCTION

CONSTRUCTION OF LL(1) PARSING TABLE

Some obvious facts about FIRST information are

3. For a rule $A \rightarrow X_1 X_2 \ldots X_r$,
   FIRST(A) = FIRST(A) $\cup$ ($\cup_k$ FIRST(X_k)),
   $1 \leq k \leq r$, provided $\epsilon \in X_i$, $1 \leq i \leq k-1$
   Thus contributions from the FIRST sets of $X_2, X_3, \ldots$ also need to be collected, only if their preceding nonterminal can derive ϵ.

How to use FIRST information in constructing the table ?

The FIRST sets for all nonterminals of the cfg is computed using Algorithm 3.1. For a rule $p \in P$ such that $p : A \rightarrow \alpha$ do the following -

1. compute FIRST(A) = FIRST(α)
2. create the entry TABLE[A,t] := $A \rightarrow \alpha$, for all t $\in$ FIRST(α)
3. set TABLE[A,a] to error , for a $\in T$, a $\notin$ FIRST(α)

The Algorithm for computing FIRST sets is given in the following and an example is given in Figure 3.10.

TOP-DOWN PARSER CONSTRUCTION

The Algorithm for computing FIRST sets is given in the following and an example is given in Figure 3.10.

FIGURE 3.10 : Constructing FIRST Information

FIRST INFORMATION

Nonterminal	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** }

TOP-DOWN PARSER CONSTRUCTION

CONSTRUCTION OF LL(1) PARSING TABLE

```
Algorithm 3.1
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
    begin
      change := false;
      for $p \in P$ such that $p$ is $A \to \alpha$ do
      begin
        newFIRST($A$) := FIRST($A$) $\cup Y$ ;
        where $Y$ is FIRST($\alpha$) computed from the definition;
        if newFIRST($A$) $\neq$ FIRST($A$) then
        begin
          change := true ; FIRST($A$) := newFIRST($A$);
        endif
      end for
    end while
  end algorithm
```

TOP-DOWN PARSER CONSTRUCTION

CONSTRUCTION OF LL(1) PARSING TABLE

FIRST Information is not enough

1. We have seen that FIRST information helps in creating some entries of the table. The point is whether it can define all the entries.
2. Using the FIRST information only (as computed in Figure 3.10), create a few rows of the parsing table and compare your answer with the actual table given in Figure 3.8.
3. The missing entries are productions of the form $A \rightarrow \epsilon$ (such productions are called ϵ*−productions*). This is natural since FIRST(ϵ) contains only { ϵ } and is missed out (there can not be a column for ϵ).
4. If a cfg has no $\epsilon -$ *productions*, then the LL(1) parsing table can be fully constructed from FIRST information alone.
5. Another information called, FOLLOW , is used to trap the $\epsilon -$ *productions*. FOLLOW(A) is defined to be the set of terminals {a}, such that a can appear immediately to the right of A in some sentential form.
6. Formal definition of FOLLOW(A) :
   FOLLOW(A) = { a | $\exists$ a derivation , $S \overset{*}{\Longrightarrow} \alpha A$ a β, for some α and β }

TOP-DOWN PARSER CONSTRUCTION

CONSTRUCTION OF LL(1) PARSING TABLE

Rules for constructing FOLLOW sets

i) For the start symbol, $\$ \in$ FOLLOW(S)

ii) If $A \rightarrow \alpha B \beta$ is a rule, then
FOLLOW(B) = FOLLOW(B) $\cup$ { FIRST(β) $- \epsilon$ }

iii) If either $A \rightarrow \alpha B$ is a rule, or if $A \rightarrow \alpha B \beta$
and $\beta \overset{*}{\Longrightarrow} \epsilon$ (i.e., $\epsilon \in$ FIRST(β)) then
FOLLOW(B) = FOLLOW(B) $\cup$ FOLLOW(A)

Example : FOLLOW sets for the grammar of Figure 3.8

- FOLLOW(*decls*) = { \$ }
- FOLLOW(*decl*) = { `var` }
- FOLLOW(*list*) = FOLLOW(*rlist*) = { : }
- FOLLOW(*type*) = { ; }

Exercise : Write an algorithm (similar to Algorithm 3.1) for constructing the FOLLOW sets of a given context free grammar.

TOP-DOWN PARSER CONSTRUCTION

CONSTRUCTION OF LL(1) PARSING TABLE

All that remains is the use of FIRST and FOLLOW sets towards the construction of the table.

- This is accomplished by creating a row of the table for each nonterminal.
- The entry for the rule $A \rightarrow \alpha$ is done as follows

1. For each a $\in$ FIRST(α), create the entry
   TABLE[A, a] = { $A \rightarrow \alpha$ }

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

3. All the remaining entries of the table are marked as syntax error.

TOP-DOWN PARSER CONSTRUCTION

CONSTRUCTION OF LL(1) PARSING TABLE

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 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*.

TOP-DOWN PARSER CONSTRUCTION

REMARKS ON LL(1) PARSER

5. 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) $\epsilon \in$ { FIRST(α) $\cap$ FIRST(β) }, multiple entries for all a $\in$ FOLLOW(A)

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

   iv) there are no other possibilities.

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

6. An ambiguous grammar, such as the if-then-else grammar, is also LL(1) ambiguous (see Figure 3.11), but the converse is not necessarily true.

FIGURE 3.11 : LL(1) Table for Dangling Else Grammar

Grammar :			
	p1,p2:	$S \longrightarrow$	i E t S S' | a
	p3,p4:	$S' \longrightarrow$	eS | ε
	p5:	$E \longrightarrow$	b

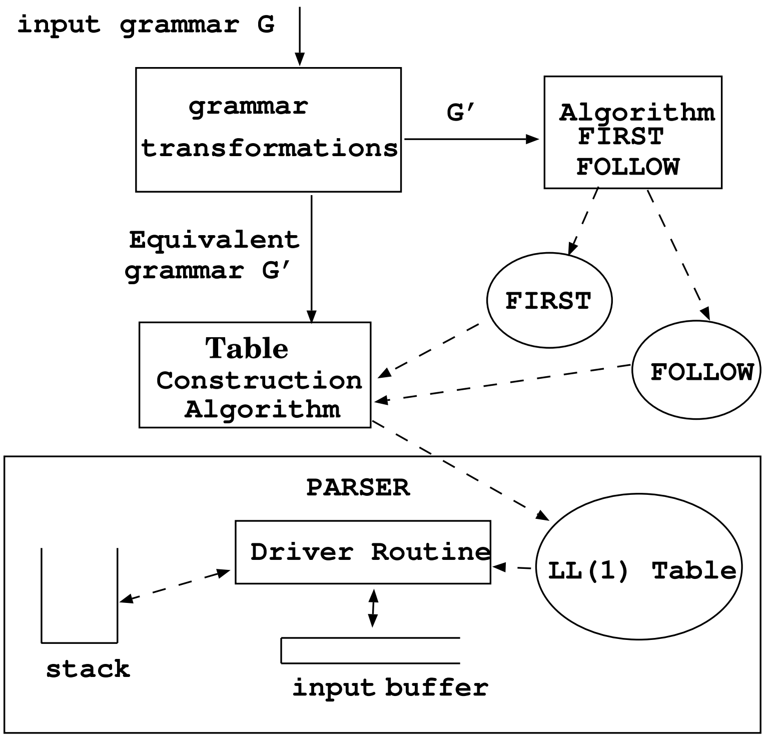
	terminal					
nonterminal	a	b	e	i	t	$
S	p2			p1		
S′			p3 p4			p4
E		p5				

TOP-DOWN PARSER CONSTRUCTION

LL(1) Parser Generation

The manual construction of LL(1) parser reveals that the entire process can be automated. Figure 3.12 shows the internal steps for generating LL(1) parser automatically from the cfg description of a PL.

FIGURE 3.12 : LL(1) Parser Generator


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