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.**

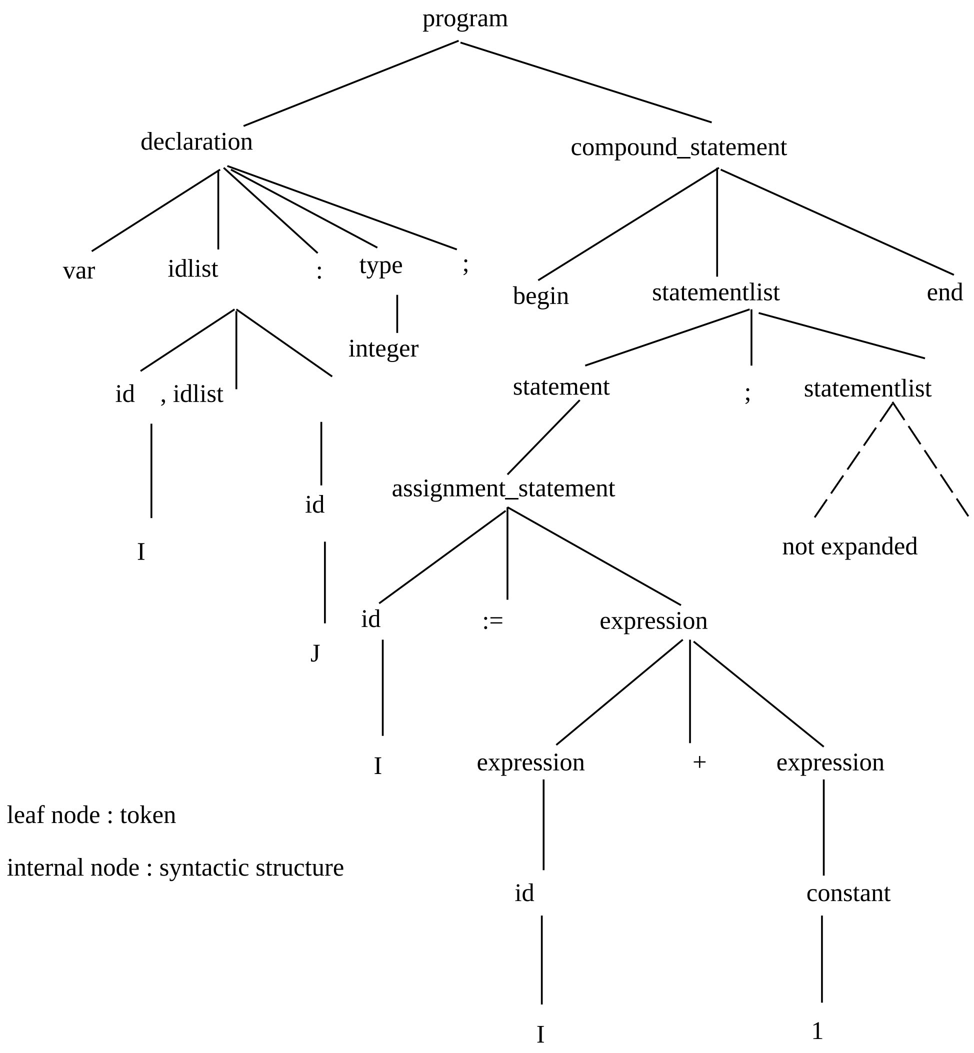
Let us try an intuitive parsing of the following program

```
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 ?

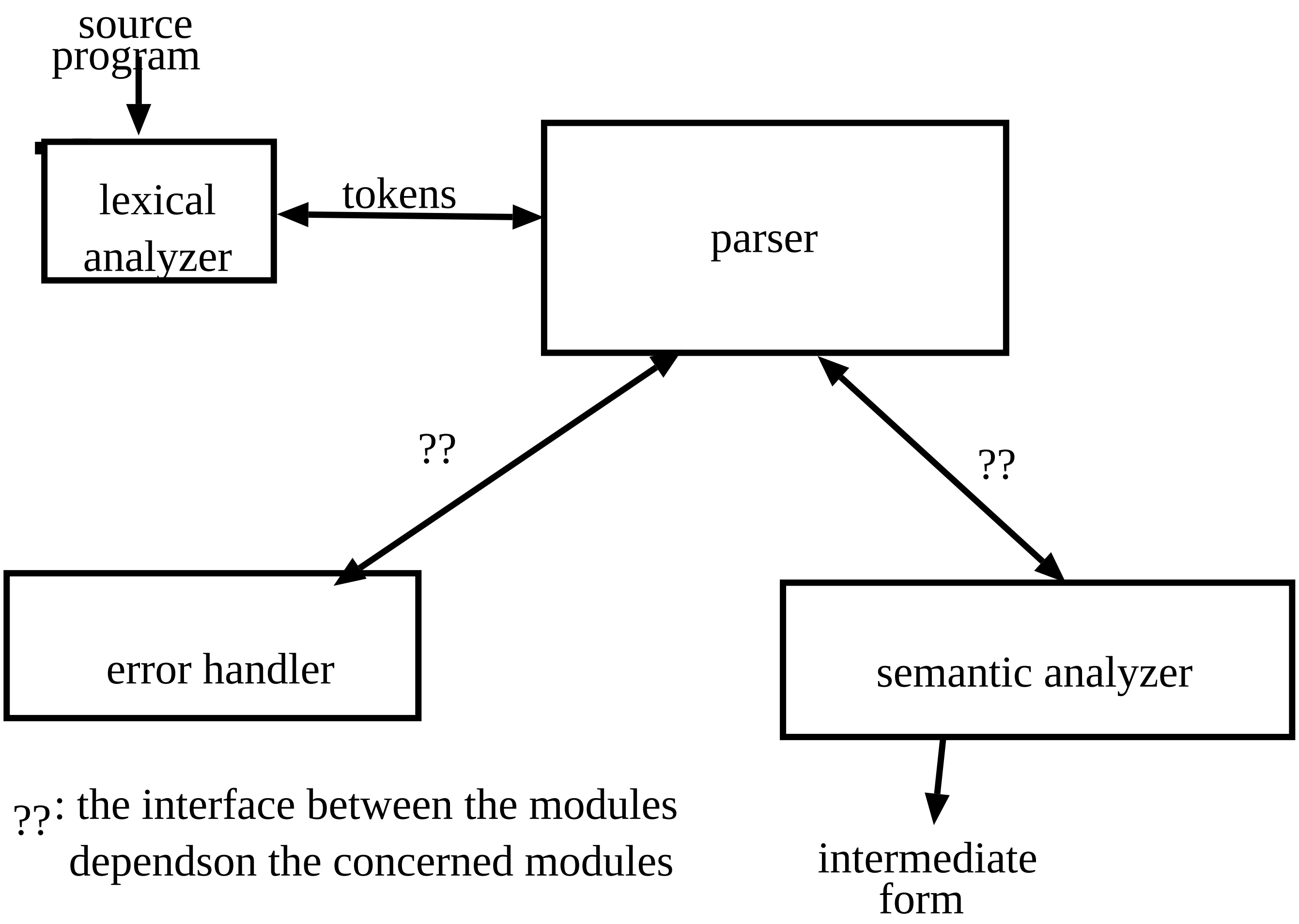
- **Till about 2 decades back, parsers (in fact all of the compiler) were written manually.**

- **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



Front end of a 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	

SYNTAX ANALYSIS

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.**

$$program \quad \rightarrow \quad \textbf{program ID } (list\ ) \textbf{ ; } decls\ compound\_statement$$

- **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**

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

A derivation is traced out as follows

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

This grammar generates terminal strings such as

ID **ID, ID** ...

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 $\stackrel{*}{\Longrightarrow}$ is used. Clearly for any string α, $\alpha \stackrel{*}{\Longrightarrow} \alpha$ is true.**

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

5. **It is interesting to note that $\Longrightarrow$, $\stackrel{*}{\Longrightarrow}$ and $\stackrel{+}{\Longrightarrow}$ 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** $L(G) = \{ \text{w} \mid S \stackrel{+}{\Longrightarrow} \text{w}, \text{w} \in T^* \}$.
   Such a string w is called a *sentence* **of** $L(G)$.

7. **A string** α, $\alpha \in (N \cup T)^*$, **such that** $S \stackrel{*}{\Longrightarrow} \alpha$, **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** $T = \{ \text{ , id } \}$; $N_1 = \{ L \}$ **and** $P_1 = \{ L \rightarrow L \text{ , id} \mid \text{id} \}$.

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

$G_3 = (T, N_3, L, P_1)$, **where** $N_3 = \{ L, L' \}$ **and** $P_3 = \{ L \rightarrow \text{id } L'$ **and** $L' \rightarrow \text{ , id } L' \mid \epsilon \}$
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 : $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Start symbol is E

Derivation of id in 3 steps $\quad E \Rightarrow T \Rightarrow F \Rightarrow id$

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

A derivation of id + id * id

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

sentential forms

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

(nonterminal chosen for expansion is underlined)

sentence

Leftmost derivation

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

$$id+\underline{T}*F \xRightarrow[lm]{} id+\underline{F}*F \xRightarrow[lm]{} id +i\,d * \underline{F} \xRightarrow[lm]{} id+id*\underline{id}$$

Rightmost derivation

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

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

BASIC CONCEPTS IN PARSING

DERIVATIONS AND PARSE TREES

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

   $A \rightarrow X_1X_2 \ldots X_n$

   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.**

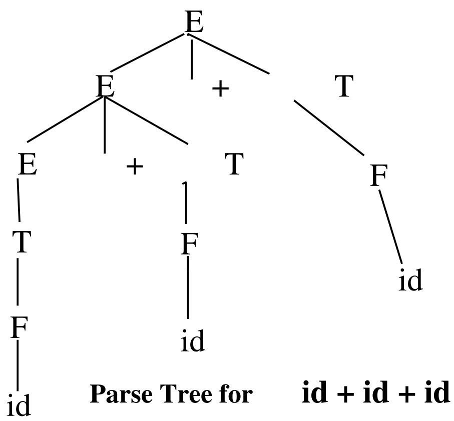
3. **Let** $\alpha_{i-1} = X_1X_2\ldots X_r \Longrightarrow X_1X_2\ldots X_{j-1}\beta X_{j+1}\ldots X_r = \alpha_i$ **and** $X_j \rightarrow Y_1Y_2\ldots Y_s$ **be the rule used.**
   - **The node (leaf) corresponding to** X_j **in the parse tree is expanded by creating** s **children for this node, 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 :

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$



Parse Tree for id + id + id

leftmost derivation

$$
\underline{E} \underset{lm}{\Rightarrow} \underline{E}+T \underset{lm}{\Rightarrow} \underline{E}+T+T \underset{lm}{\Rightarrow} \underline{T}+T+T \underset{lm}{\Rightarrow} \underline{F}+T+T \underset{lm}{\Rightarrow} id+\underline{T}+T \underset{lm}{\Rightarrow} id+\underline{F}+T \underset{lm}{\Rightarrow} id+id+\underline{T} \underset{lm}{\Rightarrow} id+id+\underline{F} \underset{lm}{\Rightarrow} id+id+id
$$

rightmost derivation

$$
\underline{E} \underset{rm}{\Rightarrow} E+\underline{T} \underset{rm}{\Rightarrow} E+\underline{F} \underset{rm}{\Rightarrow} \underline{E}+id \underset{rm}{\Rightarrow} E+\underline{T}+id \underset{rm}{\Rightarrow} E+\underline{F}+id \underset{rm}{\Rightarrow} \underline{E}+id+id \underset{rm}{\Rightarrow} \underline{T}+id+id \underset{rm}{\Rightarrow} \underline{F}+id+id \underset{rm}{\Rightarrow} id+id+id
$$

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)

   $D \rightarrow \texttt{var}\ list : type\ ;$

   $type \rightarrow \texttt{integer} \mid \texttt{real}$

   $list \rightarrow list\ ,\ \texttt{id} \mid \texttt{id}$

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

   $D \rightarrow \texttt{var}\ list : type\ ;$

   $type \rightarrow \texttt{integer} \mid \texttt{real}$

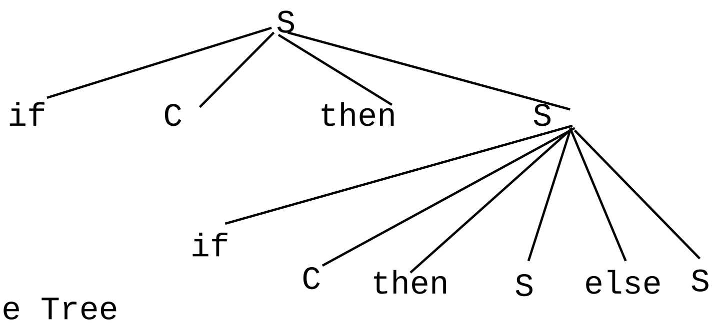
   $list \rightarrow \texttt{id}\ ,\ list \mid \texttt{id}$

FIGURE 3.5 : AN AMBIGUOUS GRAMMAR

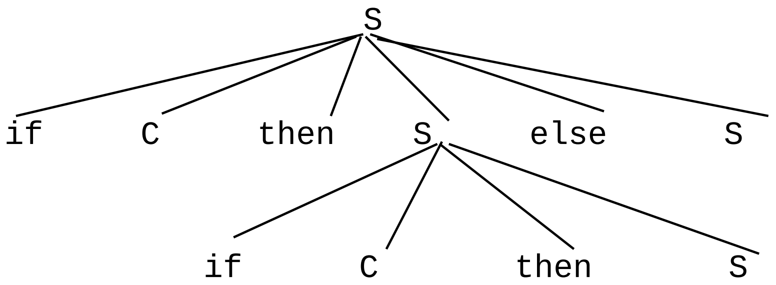
grammar is $S \longrightarrow$ `if C then S else S`

$S \longrightarrow$ `if C then S`

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



A Parse Tree



Another Parse Tree

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 :

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

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.**

   $stmt \rightarrow$ `if` exp `then` $stmt$
   $\quad\quad \mid$ `if exp then` $stmt$ `else` $stmt$ $\mid$ `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?**

(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.**

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

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**

   $A \stackrel{+}{\Longrightarrow} A\alpha$, **for some** α

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

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

REWRITING A CONTEXT FREE GRAMMAR

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

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

The transformed grammar in this is

$$\begin{array}{ll} A & \to \beta_1\ A' \mid \beta_2\ A' \mid \ldots \beta_n\ A' \\ A' & \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_m A' \mid \epsilon \end{array}$$

(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.**

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**

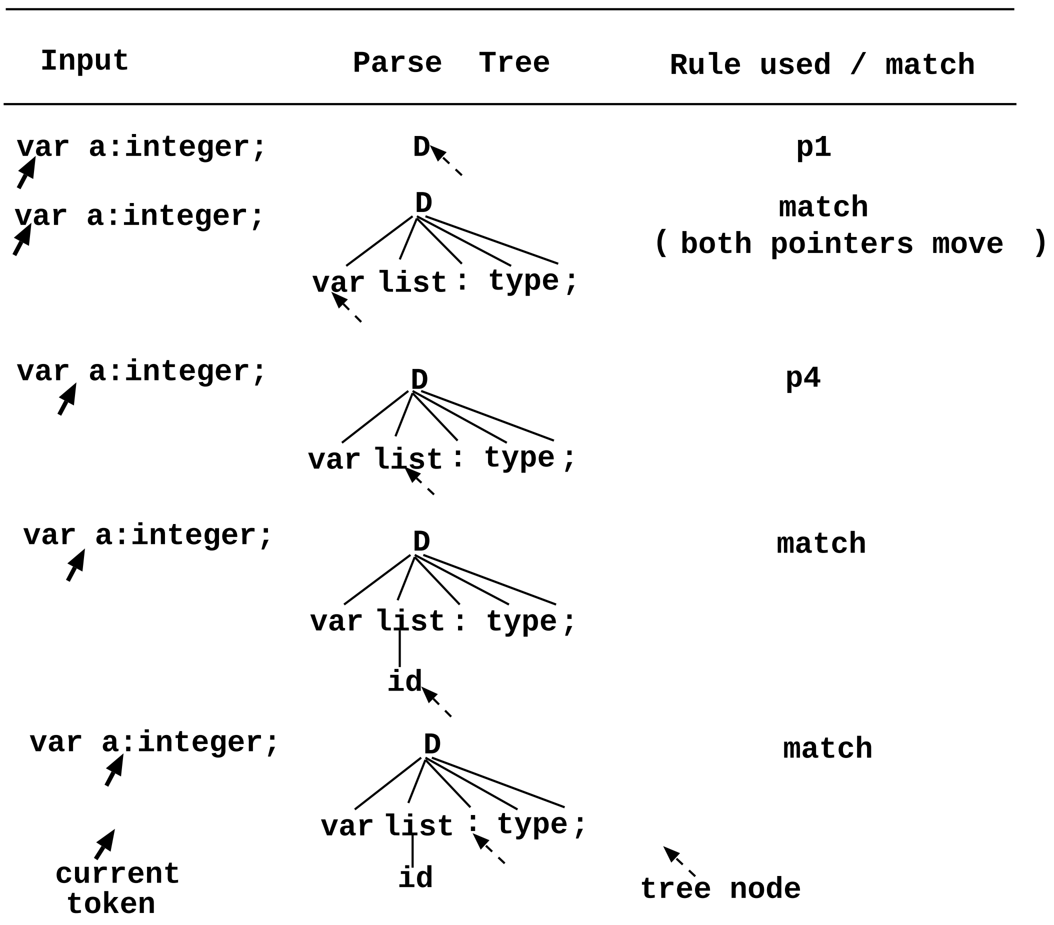
$$p_1 : D \longrightarrow \texttt{var}\ list : type;$$
$$p_2 : type \longrightarrow \texttt{integer}$$
$$p_3 : type \longrightarrow \texttt{real}$$
$$p_4 : list \longrightarrow \texttt{id}\ ,\ list$$
$$p_5 : list \longrightarrow \texttt{id}$$

is illustrated in Figure 3.6.

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

$S \rightarrow \texttt{a}\ A\ \texttt{d} \qquad A \rightarrow \texttt{bc}\ \mid \texttt{b}$

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 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, 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 $A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma$
   can be equivalently written as
   $A \rightarrow \alpha A'|\gamma$
   $A' \rightarrow \beta_1|\beta_2|\ldots|\beta_n$

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.

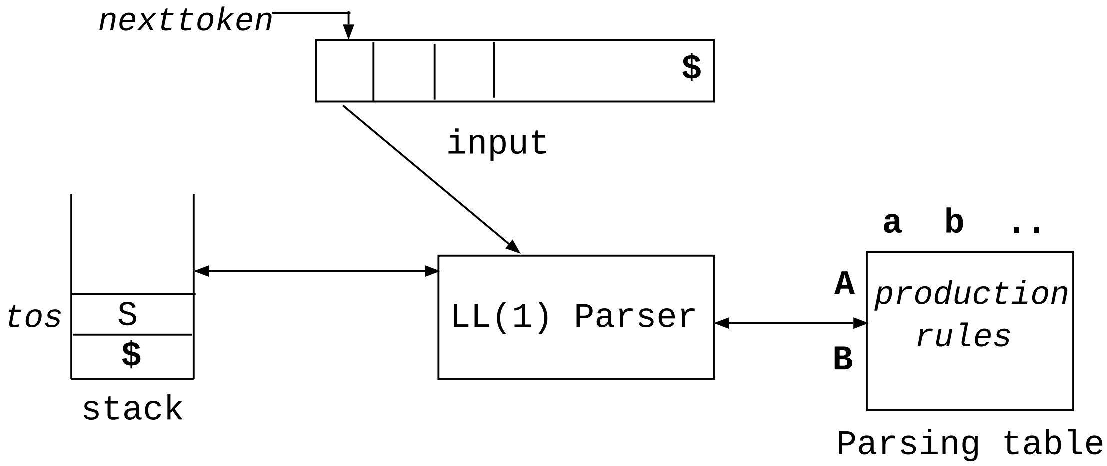
- **The components of an LL(1) parser is shown in Figure 3.7. The data structures employed by the parser are -**
  - **a stack**
  - **a table**
  - **a lookahead symbol.**

  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 → decl decls | ε

p3 : decl → var list : type ;

p4 : list → id rlist

p5 , p6 : rlist → , id rlist | ε

p7 , p8 : type → 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

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
decls	var id,id real;	use p1
decls decl	var id,id real;	use p3
decls;type:list var	var id,id real;	match
decls;type:list	id,id real;	use p4
decls;type:rlist id	id,id real;	match
decls;type rlist	,id real;	use p5
decls;type:rlist id,	,id real;	match
decls;type:rlist id	id real;	match
decls;type:rlist	real;	error

Top of the stack element is shown as symbol

nexttoken is shown as 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) = { $\mathtt{a} \mid \alpha \stackrel{*}{\Longrightarrow} \mathtt{a}\, \beta$ for some β }
    If $\alpha \stackrel{*}{\Longrightarrow} \epsilon$, then $\epsilon \in$ FIRST(α)

Some obvious facts about FIRST information are

1. For a terminal $\mathtt{a}$, FIRST($\mathtt{a}$) = { $\mathtt{a}$ }
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 $A \to X_1X_2\ldots X_r$,**
   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 $p : A \to \alpha$ do the following -

1. **compute FIRST(A) = FIRST(α)**
2. **create the entry TABLE[A,t] := $A \to \alpha$, 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 ∈ N do FIRST(A) := φ ;
    change := true;
    while change do
    begin
      change := false;
      for p ∈ P such that p is A → α do
      begin
        newFIRST(A) := FIRST(A) ∪ Y ;
        where Y is FIRST(α) computed from the definition;
        if newFIRST(A) ≠ 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 , $S \stackrel{*}{\Longrightarrow} \alpha A$ 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**$(\beta) - \epsilon$ **}**

iii) **If either** $A \rightarrow \alpha B$ **is a rule, or if** $A \rightarrow \alpha B \beta$
and $\beta \stackrel{*}{\Longrightarrow} \epsilon$ **(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 $A \to \alpha \mid \beta$ 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

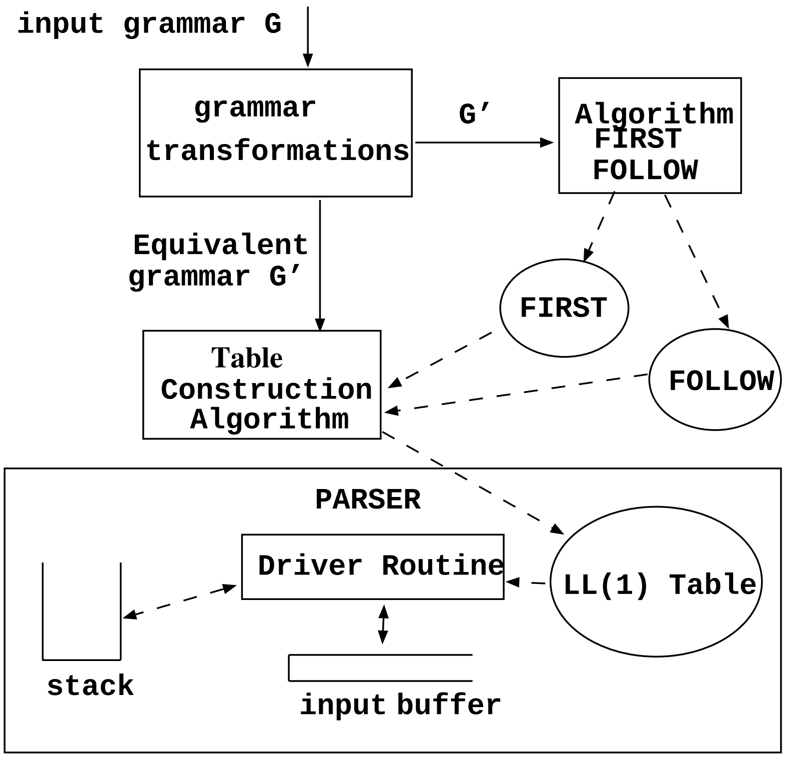
	terminal					
nonterminal	a	b	e	i	t	$
S	p2			p1		
S′			p3 p4			p4
E		p5				

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



BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

In bottom-up parsing,

1. **a parse tree is created from leaves upwards. The symbols in the input after completion of successful parse are placed at the leaf nodes of the tree.**

2. **starting from the leaves, the parser fills in the internal nodes of the unknown parse tree gradually, eventually finding the root.**

3. **the creation of an internal node involves replacing symbols from $(N \cup T)^*$ by a single nonterminal repeatedly. The task is to identify the rhs of a production rule in the tree constructed so far and replace it by the corresponding lhs of the rule. This kind of use of a production rule is called a** *reduction*.

4. **The crucial task , therefore, is to find the productions that have to be used for reduction.**

Is there a derivation that corresponds to a bottom-up construction of the parse tree ?

The construction process indicates that it has to trace out some derivation in the reverse order (provided one such exists). The following example provides the answer.

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

Example of Bottom Up Parsing

Consider the following grammar

$D \rightarrow$ `var` *list* : *type* ;

type $\rightarrow$ `integer` | `real`

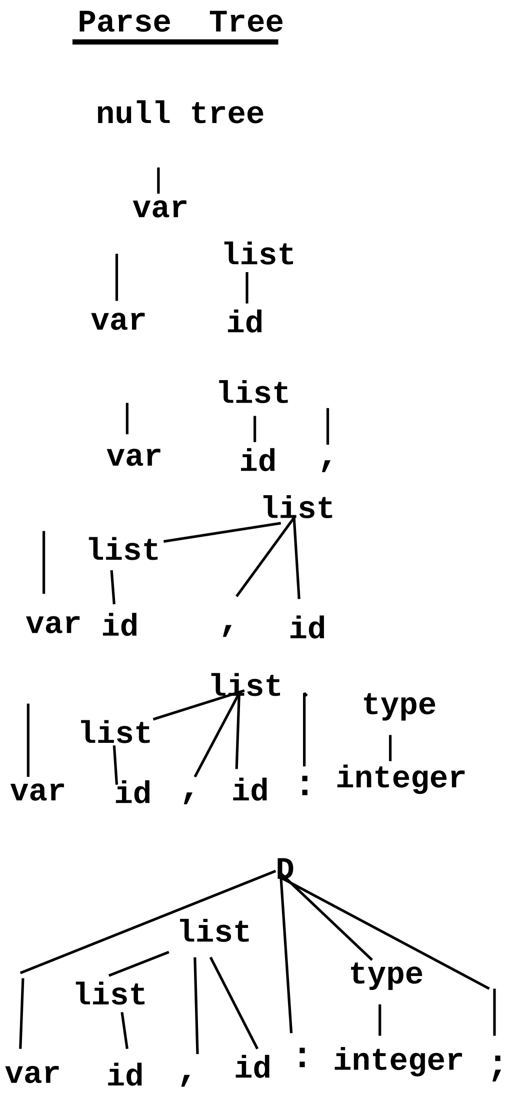
list $\rightarrow$ *list*,`id` | `id`

`var id,id : integer;` **is the input string.**

1. **The parse tree construction from the leaves is shown in Figure 3.13.**
2. **The sentential forms that were produced during the parse are**
   `var id , id : integer ;`
   `var` *list* `, id : integer ;`
   `var` *list* `: integer ;`
   `var` *list* : *type* ;
   D
3. **The sentential forms happen to be a right most derivation in the reverse order.**

FIGURE 3.13 : Example of Bottom-Up Parsing

Remaining Input	Rule used in reduction
var id,id:integer;	- - -
id,id:integer;	- - -
,id:integer;	list → id
id:integer;	- - -
:integer;	list → list,id
;	type → integer
empty	D → var list : type ;



BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

Bottom-Up parsing and rightmost derivation

Working out a rightmost derivation in reverse turns out to be a natural choice for these parsers.

1. **Right most sentential forms have the property that all symbols beyond the rightmost nonterminal are terminal symbols.**
2. **If for a string of terminals `w`, there exists a n-step** *rm-derivation* **from S, i.e.,**
   $S = \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ \ldots \ \Longrightarrow \alpha_n = w$;
   then given α_n it should be possible to construct α_{n-1} .
3. **α_{n-1} contains only one nonterminal, say A, but its position in this sentential form is not known. Examining tokens of w from left to right, one at a time, till the rhs of A is found, and then performing a reduction may give us α_{n-1} .**
4. **In general, the construction of a previous right most sentential form from a given right most sentential form is similar in spirit. The essence of bottom up parsing is to identify the rhs of a rule in the sentential forms.**
5. **It might be instructive to examine why it is not easy for a bottom-up parser to use a leftmost derivation in reverse.**

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

1. **Example : Consider the grammar for expressions, whose rules are given by**

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

   along with the input `id` $+$ `id` * `id`**.**
   Leftmost derivation of this sentence is :

   $$\begin{aligned} &E \Longrightarrow_{lm} E + T \Longrightarrow_{lm} T + T \Longrightarrow_{lm} F + T \\ &\Longrightarrow_{lm} \texttt{id} + T \Longrightarrow_{lm} \texttt{id} + T * F \Longrightarrow_{lm} \texttt{id} + F * F \\ &\Longrightarrow_{lm} \texttt{id} + \texttt{id} * F \Longrightarrow_{lm} \texttt{id} + \texttt{id} * \texttt{id} \end{aligned}$$

   (a) **There are three instances of** `id` **in the input that matches with the rhs of the rule,** $F \rightarrow$ `id`**.**
   (b) **Reducing the first** `id` **in** `id` $+$ `id` * `id` **produces** F $+$ `id` * `id` **which is not the previous sentential.**
   (c) **In order to produce the previous sentential, only the third** `id` **in** `id` $+$ `id` * `id` **should be used in the reduction.**
   (d) **The parser would have several candidates for reduction (only one correct) and it is not easy to identify the right one.**

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

Recall that the basic steps of a bottom-up parser are

1. **Reduce the input string to the start symbol by performing a series of reductions**
2. **The sentential forms produced while parsing must trace out a** *rm-derivation* **in reverse.**

There are two main considerations to carry out a reduction.

i. **to identify a** *substring* **within a** *rm-sentential* **form which matches the rhs of a rule (there may be several such)**

ii. **when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form. Such substrings, being central to bottom up parsing, have been given the name** *handle* **.**

Bottom up parsing is essentially the process of detecting handles and using them in reductions. Different bottom-up parsers use different methods for handle detection.

BOTTOM-UP PARSING

PRINCIPLES OF BOTTOM UP PARSING

Definition of a Handle

- **A *handle* of a right sentential form , say γ, is a production rule $A \rightarrow \beta$, and a position of γ such that when β is replaced by A in γ, the resulting string is the previous right sentential form in a rightmost derivation of γ.**
- **Formally, if** $S \stackrel{*}{\Longrightarrow}_{rm} \alpha\ A\ \mathtt{w} \stackrel{*}{\Longrightarrow}_{rm} \alpha\beta\ \mathtt{w}$
  then rule $A \rightarrow \beta$ in the position following α in γ is a handle of γ ($= \alpha\beta\ \mathtt{w}$).

  Only terminal symbols can appear to the right of a handle in a rightmost sentential form.

Shift-Reduce Parser

- **A parsing method that directly uses the principle of bottom up parsing outlined above is known as a *shift reduce* parser. It is a generic name for a family of bottom- up parsers that employ this strategy.**
- **A shift reduce parser requires the following data structures.**
  i. **a buffer for holding the input string to be parsed.**

BOTTOM-UP PARSING

SHIFT REDUCE PARSING

ii. a data structure for detecting handles (a stack happens to be adequate)

iii. data structure for storing and accessing the lhs and rhs of rules.

- The parser operates by applying the steps 1 and 2 repeatedly, till a situation given by either of steps 3 or 4 is found.

Step 1. Moving symbols from the input buffer onto the stack , one symbol at a time. This move is called a *shift* action.

Step 2. Checking whether a handle occurs in the stack; if a handle is detected then a reduction by an appropriate rule is performed (popping off the rhs of a rule from the stack and pushing its lhs), this move is called a *reduce* action.

Step 3. If the stack contains the start symbol only and input buffer is empty, then it announces a *successful* parse. This move is known as *accept* action.

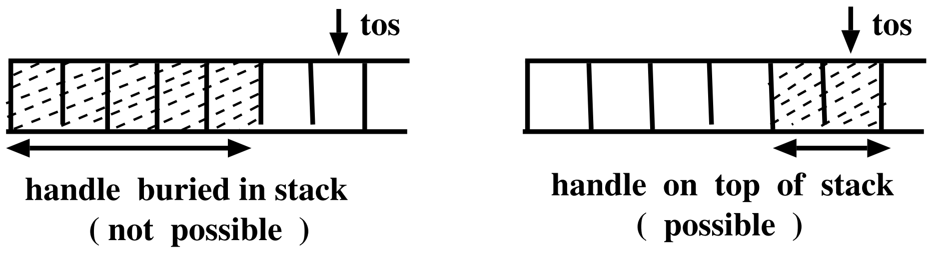
Step 4. A situation when the parser can neither shift nor reduce nor accept, it declares an error, *error* action and halts.

SHIFT REDUCE PARSING

REMARKS ON SHIFT REDUCE PARSING

1. Appending the stack contents with the unexpended input produces a rightmost sentential form.
2. Shift reduce parser as discussed, is difficult to implement , since the details of when to shift and how to detect handles are not specified clearly.
3. A property that relates to the occurrence of handles in the parser stack is exploited by shift reduce parsers. If a handle can lie anywhere within the stack then handle detection is going to be expensive since all possible contiguous substrings in the stack are potential candidates for a handle.
4. It can be proved that a handle in a shift reduce parser always occurs on top of the stack (i.e., the right end of the handle is the top of stack) and is never buried inside the stack.

FIGURE 3.15 : Occurrence of Handle in the stack



SHIFT REDUCE PARSING

Figure 3.14 : Example of Shift Reduce Parsing

grammar is

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T \,/\, F \mid F \\
F &\rightarrow P \uparrow F \mid P \\
P &\rightarrow - P \mid B \\
B &\rightarrow (E) \mid id
\end{aligned}
$$

tos

Stack	**Input**	**Parser Moves**
$	− id ↑ id / id $	shift −
$ −	id ↑ id / id $	shift id
$ − id	↑ id / id $	reduce by r12
$ − B	↑ id / id $	reduce by r10
$ − P	↑ id / id $	reduce by r9
$ P	↑ id / id $	shift ↑

Complete the rest of the parse

SHIFT REDUCE PARSING

HANDLE DETECTION IN SHIFT REDUCE PARSING

- **Example : If the stack contents happens to be αAz, with z on *tos*, then possible handles are :**
  z A z αAz and such strings
  while *substrings of* α α αA **are not handles.**

v) *Result :* **Handle in a shift reduce parser is never buried inside the stack.**

Proof Outline :

The result is easily verified for the last sentential form. Let $a_1a_2\ldots a_n$ be the sentence to be parsed and

$S \stackrel{*}{\Longrightarrow}_{rm} a_1a_2\ldots a_iAa_{i+k+1}\ldots a_n \stackrel{*}{\Longrightarrow}_{rm} a_1a_2\ldots a_n$

The last two sentential forms (say m and m-1) are shown above and $A \rightarrow a_{i+1}\ldots a_{i+k}$ is the rule used.

In this case shifting symbols upto a_{i+k} would expose the handle on top of the stack . Similar arguments hold for the other possibilities of placement of A.

general case : **Consider the stack contents at some intermediate stage of parsing , say j^{th} sentential form, and assume the handle is on (*tos*) as shown below.**

<u>stack</u>	<u>input</u>
α B	x y z

SHIFT REDUCE PARSING

Proof Outline (**Continued**)

B is currently on tos (obtained after handle pruning from the $(j+1)^{th}$ sentential form). In order to construct the $(j-1)^{th}$ sentential form, the handle in the j^{th} sentential has to be detected. There are two possible forms for the $(j-1)^{th}$ sentential.

	$(j-1)^{th}$ **sentential**	j^{th} **sentential**	**rule used**
a)	αBxAz	αBxyz	$A \rightarrow$y
b)	δAyz	αBxyz	$A \rightarrow \beta B$x

- **In (i) after shifting** y **, the handle would be found on** tos **as claimed.**
- **In (ii),** $\alpha = \delta\beta$ **and even in this situation, shifting of zero or more symbols would allow this handle to occur on** tos **and get detected.**

vi) The major implication of the above theoretical result is

1. **right end of a handle is guaranteed to occur at the top of the stack**
2. **left end of the handle would have to be found by examining as many symbols as in the rhs of the rule**
3. **searching space for a handle is reduced considerably**
4. **justifies the use of stack for detecting handles**

SHIFT REDUCE PARSING

PARSE TREE CONSTRUCTION

A parse tree can be easily constructed along with parsing by a shift reduce parser

- **the stack is extended to have another field for each entry, a pointer to a tree associated with a stack symbol. The code for tree construction is added to the basic moves of the parser.**

  (i) *shift* `a` **; create a leaf labeled** `a`**; the root of this tree is associated with the stack symbol,** `a`

  (ii) *reduce* **by** $A \rightarrow X_1X_2\ldots X_n$ **;**
  A new node labeled A **is created ; subtrees rooted at** X_1**, . . . ,** X_n **are made its children in the left to right order. The pointer for this tree rooted at** A **is stored alongwith** A **in the stack.**

  (iii) reduction by a rule, $A \rightarrow \epsilon$**, is handled by creating a tree with root labeled** A **which has a leaf labeled** ϵ **as its only child. The pointer to the root of this tree is saved alongwith** A **in the stack.**

SHIFT REDUCE PARSING

LIMITATIONS OF SHIFT REDUCE PARSER

- **What kind of grammars (languages) would not admit a shift reduce parser?**

  1. **a handle , say β, occurs at *tos*; the *nexttoken*, say a, is such that $\beta \mathtt{a} \gamma$ happens to be another handle. The parser has two options**

     (a) **reduce the handle using $A \rightarrow \beta$; in such a case the other handle , $B \rightarrow \beta \mathtt{a} \gamma$, may not get exposed for reduction and hence missed**

     (b) **ignore the handle β; shift a and continue parsing in which case reduction using $B \rightarrow \beta \mathtt{a} \gamma$ would be possible, but at the cost of missing the handle β.**
     This situation is described as a *shift reduce* conflict.

  2. **the handle β, is such that it forms the rhs of two or more rules, say $A \rightarrow \beta$ and $B \rightarrow \beta$**
     Then the parser has two or more reduce possibilities. This situation is known as a *reduce reduce* conflict.

- **How do shift reduce parsers handle such conflicts ?**
  The nexttoken could be used to prefer one move over the other. Choose shift (or reduce) in a shift reduce conflict, prefer one reduce (over others) in a reduce reduce conflict.

LR PARSERS

SOME FACTS ABOUT LR PARSERS

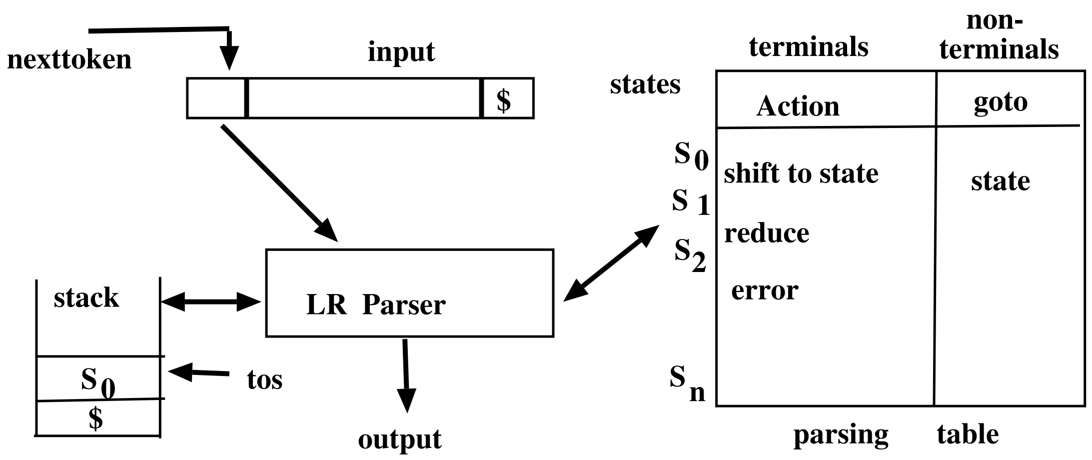
- all the parsers in the family are shift reduce parsers, the basic parsing actions are *shift*, *reduce*, *accept* and *error*
- these are the most powerful of all deterministic parsers in practice; they subsume the class of grammars that can be parsed by predictive parsers
- LR parsers can be written to recognize most of the PL constructs which can be expressed through a cfg
- the overall structure of all the parsers is the same. All are table driven , the information content in the respective tables distinguish the members of the family,
  SLR(1) $\leq$ LALR(1) $\leq$ LR(1)
  where p $\leq$ q means that q parses a larger class of grammars as compared to that parsed by p.

Structure of a LR parser

The components of a LR parser and their interaction is given in Figure 3.16. The parser apart from grammar symbols, uses extra symbols (called *states*). While both grammar and state symbols can appear on the stack, *tos* , however, always is a state.

LR PARSERS

FIGURE 3.16 : LR PARSER MODEL



Initial Configuration

Working of a LR Parser

tos	nexttoken	Action	parser action
S_j	a	s i	push a ; push S_i ; continue
	a	rj	rj : A $\longrightarrow \alpha$; $\lvert\alpha\rvert$ = r ; pop 2 r symbols ; tos = S_k ; goto[S_k , a] = S_l ; push A ; push S_l ; continue
S_j	$	acc	successful parse ; halt
S_j	a	error	error handling

LR PARSERS

EXPLANATION OF TERMS USED IN FIGURE 3.16

The parser uses two data structures prominently.

1. **a stack which contains strings of the form** $s_0X_1s_1X_2 \ldots X_ms_m$, **where** X_i **is a grammar symbol and** s_i **is a special symbol called a** *state*.

2. **a parsing table which comprises two parts, usually named as** *Action* **and** *Goto*.

   - **The Action part is a table of size** $n \times m$ **where n is the total number of states of the parser and** $m = |T|$ **(including \$). The possible entries are**
     i) s_i **which means shift to state i**
     ii) r_j **which stands for reduce by the** j^{th} **rule,**
     iii) *accept*
     iv) *error*

   - **The Goto part of the parsing table is of size** $n \times p$, **where** $p = |N|$. **The only interesting entries are state symbols.**

SLR(1) PARSER

CONFIGURATION OF A LR PARSER

The LR parser is a driver routine which

- **i) initializes the stack with the** *start* **state and calls scanner to get a token (***nexttoken***).**
- **ii) For any configuration, as indicated by (***tos , nexttoken***), it consults the parsing table and performs the action specified there.**
- **iii) The goto part of the table is used only after a reduction.**
- **iv) The parsing continues till either an error or accept entry is encountered.**

Simple LR(1) Parser

- **We start by examining the parser of the family which is known as Simple LR(1) or SLR(1) parser. SLR(1) Parsing Table for an expression grammar is given in Figure 3.17.**
- **The working of this parser for a sample input is shown in Figure 3.18.**

SLR(1) PARSER

FIGURE 3.17 : SLR(1) PARSING TABLE

LR Parsing Table for an expression grammar

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

	Action						***Goto***		
state	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			

FIGURE 3.18 : WORKING OF A SLR(1) PARSER

Parsing the Input * a = * * b using LR Parsing Table of Fig 3.17

Stack	Input	Action table	Goto table
$ 0	* a = * * b $	[0 , *] = s4	—
$ 0 * 4	a = * * b $	[4 , id] = s5	—
$ 0 *4 id 5	= * * b $	[5 , =] = r 4	[4 , L] = 8
$ 0 *4 L 8	= * * b $	[8 , =] = r 5	[4 , R] = 7
$ 0 *4 R 7	= * * b $	[7 , =] = r 3	[0, L] = 2
$ 0 L 2	= * * b $	[2 , =] = s 6	—
$ 0 L 2 = 6	* * b $	[6 , *] = s12	—
$ 0 L 2 = 6 * 12	* b $	[12 , *] = s12	—
$ 0L2=6*12*12	b $	[12 , id] = s11	—
$ 0L2=6*12*12id 11	$	[11 , $] = r 4	[12, L] = 10
$ 0L2=6*12*12L10	$	[10 , $] = r 5	[12 , R] = 13
$ 0L2=6*12*12R13	$	[13 , $] = r 3	[12 , L] = 10
$ 0 L 2 = 6 *12 L 10	$	[10 , $] = r 5	[12 , R] =13
$ 0L 2 = 6 *12 R 13	$	[13, $] = r 3	[6, L] = 10
$ 0 L 2 = 6 L 10	$	[10, $] = r5	[6, R] = 9
$ 0 L 2 = 6 R 9	$	[9, $] = r1	[0, S] = 1
$ 0 S 1	$	[1, $] = acc	—

SLR(1) PARSER

CONFIGURATION OF A LR PARSER

- **A *configuration* of a LR parser is defined by a tuple, (stack contents , unexpended part of input).**
- **Initial configuration is $(s_0, a_1a_2 \dots a_n\$)$, where s_0 is a designated *start* state and the second component is the entire sentence to be parsed.**
- **Let an intermediate configuration be given by $(s_0X_1s_1 \dots X_is_i \ , \ a_ja_{j+1} \dots a_n\$)$, then resulting configuration**

  i) after a shift action is given by
  $(s_0X_1s_1 \dots X_is_ia_js_k, a_{j+1} \dots a_n\$)$
  provided $Action[s_i, a_j] = s_k$; both *stack* and *nexttoken* change after a shift.

  ii) after a reduce action is given by
  $(s_0X_1s_1 \dots X_{i-r}As, a_j \dots a_n\$)$
  where $Action[s_i, a_j] = r_l$; rule $p_l : \ A \rightarrow \beta$, β has r grammar symbols and $goto(s_{i-r}, A) = s$. Only the *stack* changes here.

SLR(1) PARSER CONSTRUCTION

We now study SLR(1) parsing table construction. The material would be covered in the following order.

1. **The relevant definitions are introduced first.**
2. **The construction process is then explained in full detail.**
3. **Theory of LR parsing which provides the basis for the construction process and also shows its correctness is addressed last.**

DEFINITION OF LR(0) ITEM AND RELATED TERMS

- *LR(0) item* **: An LR(0) item for a grammar G is a production rule of G with the symbol $\bullet$ (read as *dot* or *bullet*) inserted at some position in the rhs of the rule.**
- **Example of LR(0) items : For the rule given below**

  $decls \rightarrow decls\ decl$

  the possible LR(0) items are :

  $I_1 : decls \rightarrow \bullet decls\ decl$

  $I_2 : decls \rightarrow decls\ \bullet decl$

  $I_3 : decls \rightarrow decls\ decl\bullet$

  The rule $decls \rightarrow\ \epsilon$ has only one LR(0) item,

  $I_4 : decls \rightarrow\ \bullet$

SLR(1) PARSER CONSTRUCTION

DEFINITION OF LR(0) ITEM AND RELATED TERMS

- *Incomplete and Complete LR(0) items* : **An LR(0) item is complete if the $\bullet$ is the last symbol in the rhs, else it is an incomplete item.**
  For every rule, $A \rightarrow\ \alpha$, $\alpha \neq\ \epsilon$ there is only one complete item, $A \rightarrow\ \alpha\bullet$ but as many incomplete items as there are grammar symbols in the rhs.
  Example : I_3 and I_4 are complete items and I_1 and I_2 are incomplete items.

- *Augmented Grammar* : **From the grammar, $G = (N, T, P, S)$, we create a new grammar $G' = (N', T, P', S')$, where $N' = N \cup \{S'\}$, S' is the start symbol of G' and $P' = P \cup \{S' \rightarrow S\}$**
  Note that G' is equivalent to G.

- *Kernel and Nonkernel items* : **An LR(0) item is called a kernel item, if the dot is not at the left end. However the item $S' \rightarrow\ \bullet S$ is an exception and is defined to be a kernel item.**

  An LR(0) item which has dot at the left end is called a nonkernel item.

  Example : I_1, I_2, I_3 are all kernel items and I_4 is a nonkernel item.

SLR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(0) ITEMS

- *Functions closure and goto* : **Two functions** *closure* **and** *goto* **are defined which are used to create a set of items from a given item.**

  Let U be the collection of all LR(0) items of a cfg G. The *closure* **function has the form** $f : U \rightarrow 2^U$ **and is constructed as follows :**

  i) $closure(\mathbf{I}) = \{\ \mathbf{I}\ \}$**, for** $\mathbf{I} \in \mathbf{U}$

  ii) If $A \rightarrow \alpha \bullet B\beta \in closure(\mathbf{I})$**, then for every rule of the form** $B \rightarrow \eta$**, the item** $B \rightarrow \bullet\eta$ **is added to (if it is not already there)** $closure(\mathbf{I})$**.**

  iii) Apply step (ii) above repeatedly till no more new items can be added to $closure(\mathbf{I})$**.**

  $closure(\mathbf{I}) \neq \{\mathbf{I}\}$ **only when I is an item in which a nonterminal immediately follows the dot.**

- **The** *goto* **function has the form** $f : U \times X \rightarrow 2^U$**, where** X **is a grammar symbol.**

SLR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(0) ITEMS

- *goto* is defined using *closure* as follows
  $goto(A \rightarrow \alpha \bullet X\beta\ ,X) = closure(A \rightarrow \alpha X \bullet \beta\ )$.
  Clearly $goto(\mathbf{I},X)$ for a complete item **I** is Φ, for all X.

- Example of *closure* and *goto* : Consider $A \rightarrow A\ \mathtt{a} \mid \mathtt{b}$
  (a) $closure(A \rightarrow \bullet A\ \mathtt{a}) = \{\ A \rightarrow \bullet A\ \mathtt{a}\ \}$, by rule (i).
  The item $A \rightarrow \bullet\ \mathtt{b}$ is added to it by rule (ii).
  $closure(A \rightarrow \bullet A\ \mathtt{a}) = \{\ A \rightarrow \bullet A\ \mathtt{a}, A \rightarrow \bullet\ \mathtt{b}\ \}$ after step(iii).

  (b) $goto(A \rightarrow \bullet A\ \mathtt{a},\ A) = closure(A \rightarrow\ A \bullet\ \mathtt{a})$
  $= \{\ A \rightarrow\ A \bullet\ \mathtt{a}\ \}$

- The functions *closure* and *goto* can be extended to a set S of LR(0) items by appropriate generalizations :
  $closure(\mathbf{S}) = \cup_{\mathtt{I} \in \mathtt{S}}\ \{closure(\mathbf{I})\ \}$;
  $goto(\mathbf{S},X) = \cup_{\mathtt{I} \in \mathtt{S}}\ \{goto(\mathbf{I},X)\ \}$
  Algorithm 3.2 gives a pseudo code for computing these functions.

Given a cfg, the collection U of all LR(0) items is well defined. A particular collection C of sets of items i.e., $C \in 2^U$, is of specific interest and is called the *canonical collection* of LR(0) items. This collection is constructed using *closure* and *goto* as given in Algorithm 3.2.

SLR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(0) ITEMS

```
Algorithm 3.2
procedure items(G', C);
    begin
        i := 0;  I_0 := { closure( S' → •S )};   C := I_i;
        repeat
            for  each set of items I_i in C and each grammar
            symbol X, such that goto(I_i,X) is not empty
            and ∉ C, do
            i := i + 1 ;  I_i := goto(I_i,X); C := C ∪ I_i
        until no more sets of items can be added to C
    end
procedure closure(I); { I is a set of items }
    begin
        repeat
            for each item A → α • Bβ ∈ I
            and each rule B → γ such that B → •γ is not in I
            do add B → •γ to I;
        until no more items can be added to I ;
        return I ;
    end
```

SLR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(0) ITEMS

```
function goto(I, X)
    begin
        goto_items := φ ;
        for  each item A → α • Xβ ∈ I,
              such that A → αX • β is not in goto_items
              do add A → αX • β to goto_items
        goto := closure(goto_items);
        return goto ;
    end
```

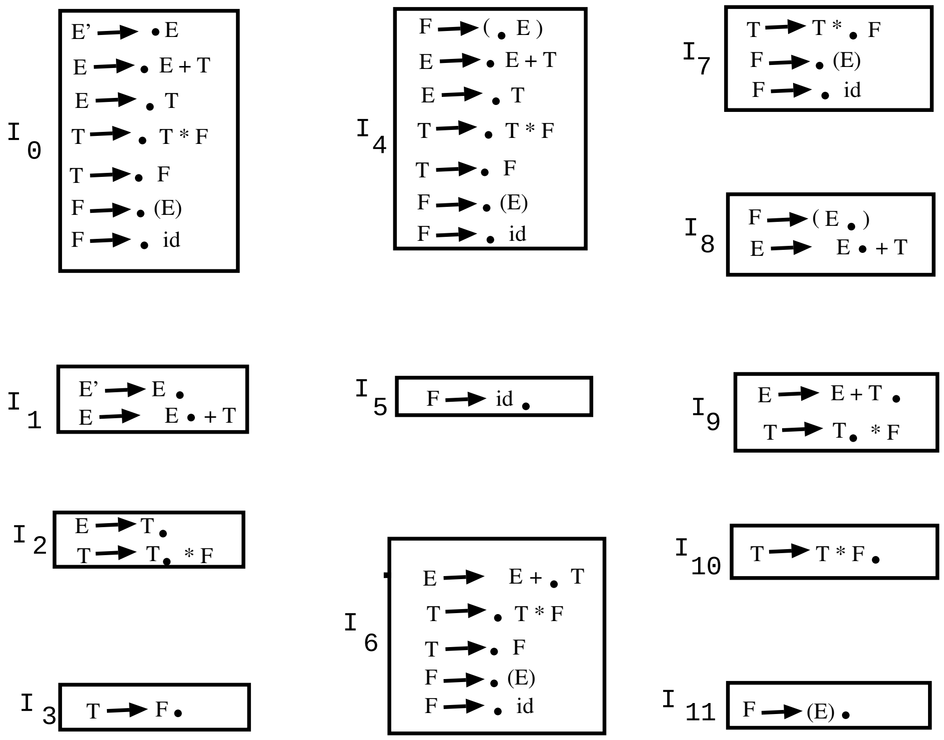
Illustration of construction of canonical collection

- **Consider the expression grammar of Figure 3.17 . I_0 is closure($E' \rightarrow \bullet E$). The items $E' \rightarrow \bullet E$, $E \rightarrow \bullet E+T$ and $E \rightarrow \bullet T$ are added to I_0. The last one in turn causes the addition of $T \rightarrow \bullet T * F$ and $T \rightarrow \bullet F$. The item $T \rightarrow \bullet F$ leads to the addition of $F \rightarrow \bullet(E)$ and $F \rightarrow \bullet$ `id`. No more items can be added and the collection I_0 is complete (see Figure 3.19)**

- **I_0 has items having $\bullet\alpha$ combination for $\alpha =$ { E , T, F, (, `id` }. $goto(I_0, \alpha)$ is defined for all these values of α which in turn give rise to different collections as shown in Figure 3.19.**

SLR(1) PARSER CONSTRUCTION

FIGURE 3.19 : Canonical Collection of LR(0) items



SLR(1) PARSER CONSTRUCTION

CONSTRUCTION OF SLR(1) PARSING TABLE

From the collection C constructed using Algorithm 3.2, one can directly construct a SLR(1) parsing table. The procedure is described in Algorithm 3.3.

Algorithm 3.3 :
Input : Grammar G
Output : Action and goto part of the parsing table

1. **From the input grammar G, construct an equivalent augmented grammar G'.**
2. **Use the algorithm for constructing FOLLOW sets to compute FOLLOW(A), $\forall\ A\ \in\ N'$.**
3. **Call procedure items(G',C) to get the desired canonical collection $C = \{I_0, I_1, \ldots, I_n\}$.**
4. **Choose as many state symbols as the cardinality of C. We use numbers 0 through n to represent states.**
5. **The set I_i and its constituent items define the entries for state i of the Action table and the Goto table.**
   For each $I_i,\ 0 \leq i \leq n$ do steps 5.1 through 5.6 given below.

   5.1 if $A \rightarrow \alpha \bullet$ a $\beta \in I_i$ and goto(I_i,a) $= I_j$, then
   Action[i,a] $= s_j$, i.e., shift to state j

SLR(1) PARSER CONSTRUCTION

CONSTRUCTION OF SLR(1) PARSING TABLE

5.2 If $A \rightarrow \alpha \bullet \in I_i$, then
Action[i, a] $= r_j$, $\forall$ a $\in$ FOLLOW(A)
which means reduce by the j^{th} rule $A \rightarrow \alpha$.

5.3 If I_i happens to contain the item $S' \rightarrow S\bullet$, then
Action[i,\$] = accept

5.4 All remaining entries of state i in the Action table are marked as `error`

5.5 For nonterminals A , such that $goto(I_i, A) = I_j$ create
Goto[i,A] = j

5.6 The remaining entries in the Goto table are marked as error.

NOTE : The Goto entries of the state i of the parsing table i are decided by the $goto(I_i, A)$ states where A is a nonterminal.

The SLR(1) table for the canonical collection of Figure 3.19 is partly constructed in Figure 3.20.
Remarks : 1. *initial or start state* **of the parser is the state corresponding to the set I_0 which contains the kernel item $S' \rightarrow \bullet S$**
2. The parsing table is conflict free since there is no multiple entry (shift-reduce and/or reduce-reduce) in it.

FIGURE 3.20 : Illustration of Algorithm 3.3

State i	Item with $.t$ in I_i	GOTO (I_i, t)	Complete item	Action Table
0	$F \rightarrow .\, id$ $F \rightarrow .\, (E)$	I_5 I_4	none	shift 5 shift 4
1	$E \rightarrow E .\, + T$	I_6	$E' \rightarrow E .$	shift 6 acc on $

State i	Item with $.A$ in I_i	GOTO (I_i, A)	goto Table
0	A = E A = T A = F	I_1 I_2 I_3	1 2 3
1	none		

SLR(1) PARSER CONSTRUCTION

SLR(1) GRAMMAR AND PARSER

A grammar for which there is a conflict free SLR(1) parsing table is called a SLR(1) grammar and a parser which uses such a table is known as SLR(1) parser.

- **How to detect shift-reduce conflicts and reduce-reduce conflicts in SLR(1) parser ?**

- **Step 5 of Algorithm 3.3 contains all the information. A shift- reduce conflict is detected when a state has**
  (a) a complete item of the form $A \rightarrow \alpha\bullet$ **with** $\mathsf{a} \in$ **FOLLOW**(A) **and also**
  (b) an incomplete item of the form $B \rightarrow \beta\bullet\ \mathsf{a}\ \gamma$

- **A reduce-reduce conflict is noticed when a state has two or more complete items of the form**
  (a) $A \rightarrow \alpha\bullet$
  (b) $B \rightarrow \beta\bullet$ **, and**
  (c) FOLLOW$(A) \cap$ **FOLLOW**$(B) \neq \Phi$

Having seen how to construct SLR(1) parsing table manually, we address the conceptual aspects of this method in order to answer questions of the following kind.

1. **What are the state symbols used by the parser and what information do these contain ?**

VIABLE PREFIXES AND VALID ITEMS

2. **Where exactly is handle detection taking place in the parser?**
3. **Why is FOLLOW information used to create the reduce entries in the action table ?**

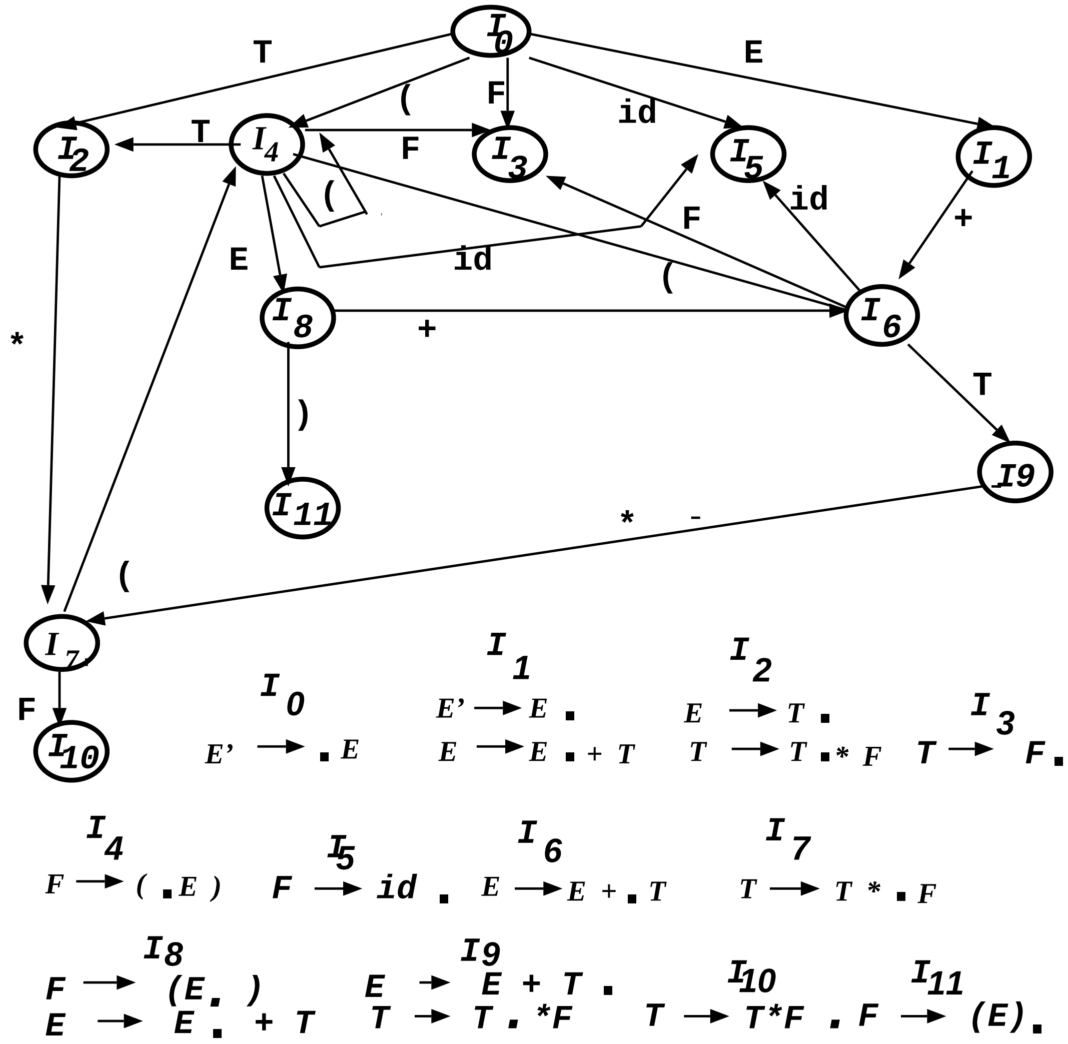
The canonical collection of LR(0) items can also be represented by a directed labeled graph.

- **A node labeled I_i is constructed for each member of C.**
- **For every nonempty goto(I_i, X) $= I_j$, a directed edge (I_i, I_j) is added labeled with X.**
- **The graph is a deterministic finite automaton if the node labeled I_0 is treated as the *start* state and all other nodes are made final states.**
- **The finite automaton associated with the collection of items given in Figure 3.19 is shown in Figure 3.21 . The crucial question is to understand what this automata recognizes. The following terms are defined in order to precisely characterize these string of symbols.**

VIABLE PREFIX

A *viable prefix* of a grammar is defined to be the prefixes of right sentential forms that do not contain any symbols to the right of a handle.

FIGURE 3.21 : LR AUTOMATON FOR EXPRESSION GRAMMAR



Only the kernel items of each state is shown in the above

THEORY OF LR PARSING

VIABLE PREFIX AND VALID ITEMS

1. **By adding terminal symbols to viable prefixes, rightmost sentential forms can be constructed.**
2. **Viable prefixes are precisely the set of symbols that can ever appear on the stack of a shift reduce parser**
3. **A viable prefix either contains a handle or contains parts of one or more handles.**
4. **Given a viable prefix, if one could identify the set of potential handles associated with it then a recognizer for viable prefixes would also recognize handles.**

The significance of LR(0) item can now be understood. A complete item corresponds to a handle while an incomplete item indicates the part of a handle seen so far.
A LR(0) item $A \to \beta_1 \bullet \beta_2$ **is defined to be** *valid* **for a viable prefix,** $\alpha\beta_1$**, provided** $S \overset{*}{\Longrightarrow}_{rm} \alpha A \texttt{ w} \Longrightarrow_{rm} \alpha\beta_1\beta_2 \texttt{ w}$

1. **It is interesting to note that in above, if** $\beta_2 = B\gamma$ **and** $B \to \delta$**, then** $B \to \bullet\delta$ **is also a valid item for this viable prefix.**
2. **There could be several distinct items which are valid for same viable prefix** γ**.**

VIABLE PREFIXES AND VALID ITEMS

3. **A particular item may be valid for many distinct viable prefixes.**
4. **The parser would halt when the viable prefix S is seen, in the item sense, it indicates a move from $\bullet S$ to $S\bullet$. This is the reason for augmenting the input grammar.**

- **Example : For the LR-automaton given in Figure 3.21, these concepts and their relationships are explained in the following.**
- **Consider the path { I_0, I_4, I_8, I_6 } which has the label (E + along it. The items that are valid for the viable prefix, (E + are determined below.**

  1. $E' \Longrightarrow_{rm} E \Longrightarrow T \Longrightarrow F \Longrightarrow (E) \Longrightarrow (E+T)$ **shows that** $E \rightarrow E + \bullet T$ **is a valid item**

  2. $E' \Longrightarrow E \Longrightarrow T \Longrightarrow F \Longrightarrow (E) \Longrightarrow (E+T) \Longrightarrow (E+T*F)$ **shows that** $T \rightarrow \bullet T * F$ **is also a valid item.**

  3. $E' \overset{*}{\Longrightarrow}_{rm} (E+T) \Longrightarrow (E+F)$ **shows that** $T \rightarrow \bullet F$ **is another such item.**

  4. $E' \overset{*}{\Longrightarrow}_{rm} (E+T) \Longrightarrow (E+F) \Longrightarrow (E+(E))$ **shows that** $F \rightarrow \bullet (E)$ **is a valid item.**

  5. $E' \overset{*}{\Longrightarrow}_{rm} (E+F) \Longrightarrow (E+\texttt{id})$ **shows that** $F \rightarrow \bullet\ \texttt{id}$ **is a valid item.**

  It should be noted that are no other valid items for this viable prefix.

THEORY OF LR PARSING

VIABLE PREFIXES AND VALID ITEMS

- **Given a LR(0) item, say $T \rightarrow T \bullet *F$, there may be several viable prefixes for which it is valid.**

  1. $E' \Longrightarrow_{rm} E \Longrightarrow T \Longrightarrow T * F$ **shows that this item is valid for the viable prefix** T **.**

  2. $E' \Longrightarrow E \Longrightarrow T \Longrightarrow F \Longrightarrow (E) \Longrightarrow (T) \Longrightarrow (T * F)$ **shows that it is also valid for** (T **.**

  3. $E' \Longrightarrow E \Longrightarrow T \Longrightarrow T * F \Longrightarrow T * (E) \Longrightarrow T * (T) \Longrightarrow T * (T * F)$ **shows that it is valid also for** T * (T **.**

  4. $E' \Longrightarrow E \Longrightarrow E + T \Longrightarrow E + T * F$ **shows validity for** $E + T$ **. There may be several other viable prefixes for which this item is valid.**

- *THEOREM :* **The set of all valid items for a viable prefix γ is exactly the set of items reached from the start state along a path labeled γ in the DFA that can be constructed from the canonical collection of sets of items, with the transitions given by** *goto***.**

THEORY OF LR PARSING

VIABLE PREFIXES AND VALID ITEMS

- **The theorem stated without proof above is a key result in LR Parsing. It provides the basis for the correctness of the construction process we learnt earlier.**

- **An LR parser does not scan the entire stack to determine when and which handle appears on top of stack (compare with shift reduce parser).**

- **The state symbol on top of stack provides all the information that is present in the stack.**

- **In a state which contains a complete item a reduction is called for. However, the lookahead symbols for which the reduction should be applied is not obvious.**

- **In SLR(1) parser the FOLLOW information is used to guide reductions.**

CANONICAL LR(1) PARSER

Canonical LR(1) parser is the next parser that we study. In order to motivate the need for having another LR parser, the limitations of SLR(1) parser are pointed out.

FOLLOW information is imprecise

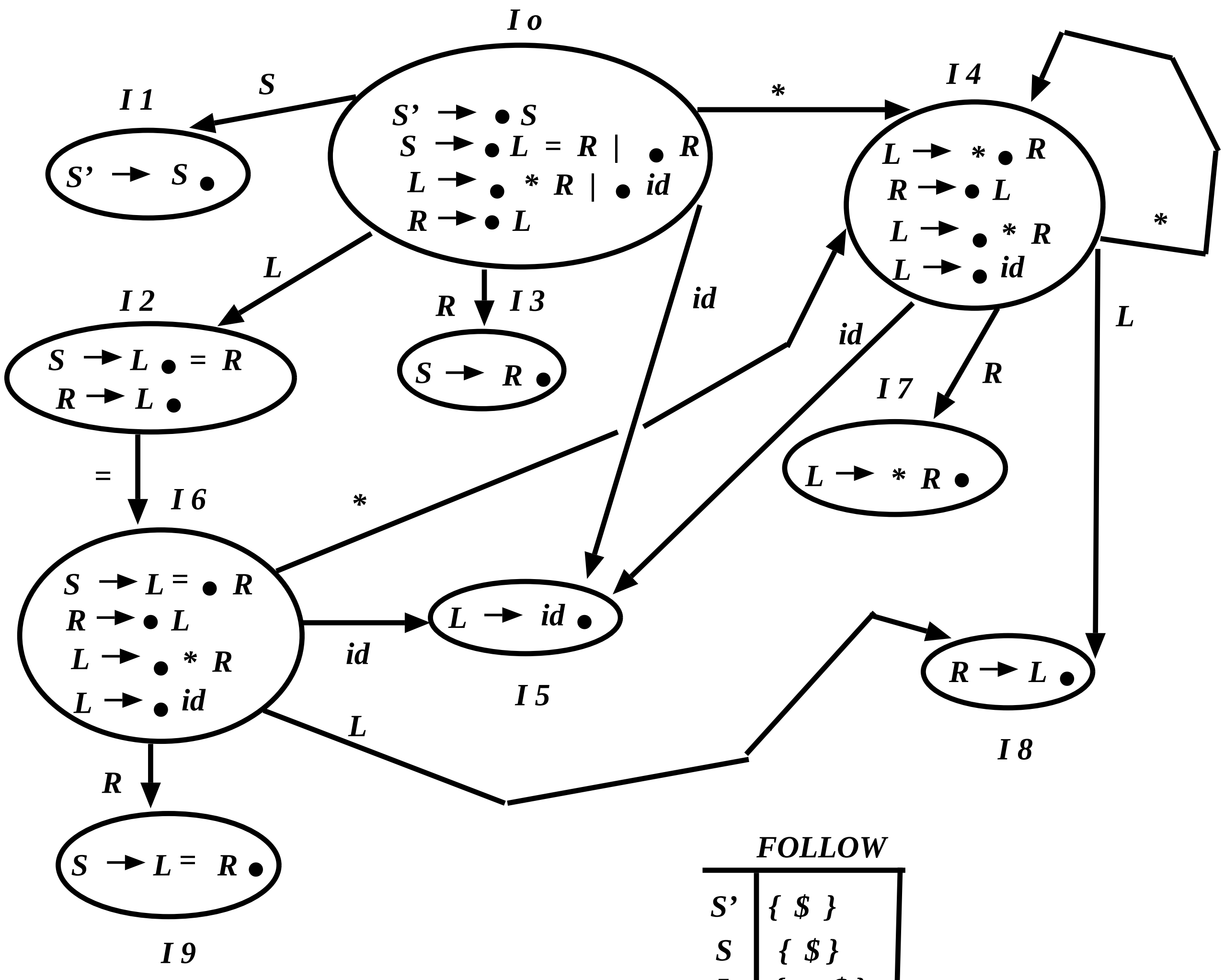
- **An example to illustrate this fact is the following grammar, which can be seen to be SLR(1) unambiguous.**

$$S \rightarrow L \ = \ R \mid R$$
$$L \rightarrow \ * \ R \mid \texttt{id}$$
$$R \rightarrow \ L$$

- **The SLR automaton and the parsing table for this grammar are given in the Figures 3.22 and 3.23 respectively.**
- **Observe that** $\{=\} \in$ **FOLLOW**(R)**, and** $R \rightarrow L\bullet$ **is an item in state 2. There is another item,** $S \rightarrow L\bullet = R$ **in the same state**
- **Note that state 2 recognizes the viable prefix** L **and the shift entry is justifiable.**
- **How about the reduce entry ? After seeing** L**, if we reduce it to** R**, with the expectation of = to follow, there must exist a viable prefix** $R = \ldots$ **and hence a right sentential form of the form** $R =$ **z. It can be shown that such is not possible. The problem seems to be with our FOLLOW information.**

LR(1) PARSER CONSTRUCTION

FIGURE 3.22 : SLR AUTOMATON FOR A GRAMMAR



FOLLOW

S'	{ $ }
S	{ $ }
L	{ = $ }
R	{ = $ }

FIGURE 3.23 : SLR PARSING TABLE

Grammar :

S′ → S
S → L = R | R
L → * R | id
R → L

	Action				Goto		
state	id	*	=	$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6 r5	r5	-	-	-
3	-	-	-	r2	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r4	r4	-	-	-
6	s5	s4	-	-	-	8	9
7	-	-	r3	r3	-	-	-
8	-	-	r5	r5	-	-	-
9	-	-	-	r1	-	-	-

LR(1) PARSER CONSTRUCTION

LR(1) ITEM AND LOOKAHEAD SYMBOLS

What is wrong with FOLLOW ?

- **This information is not correct for state 2.**
- **The sentential forms that permit = to follow R are of the form $*L\ldots$ and taken care of in the state 8 of the parser.**
- **The context in state 2 is different (viable prefix is L), and use of FOLLOW constrains the parser.**

Given an item and a state the need is to identify the terminal symbols that can actually follow the lhs nonterminal.

An item of the form, $A \rightarrow \alpha \bullet \beta$, a where the first component is a LR(0) item and the second component is a set of terminal symbols is called a LR(1) item. The second component is known as *lookahead* symbol. The value 1 indicates that the length of lookahead is 1.

- **The lookahead symbol is used in a reduce operation only. For an item of the form, $A \rightarrow \alpha \bullet \beta$, a if β is not ϵ, the lookahead has no effect. But for an item, $A \rightarrow \alpha\bullet$, a the reduction is applied only if the *nexttoken* is a.**

LR(1) PARSER CONSTRUCTION

VALID LR(1) ITEM

- **An LR(1) item $A \to \alpha \bullet \beta$, $\mathtt{a}$ is valid for a viable prefix γ, if there is a derivation**
  $S \overset{*}{\Longrightarrow}_{rm} \delta A\ \mathtt{w} \Longrightarrow_{rm} \delta\alpha\beta\ \mathtt{w}$, **where**
  i) $\gamma = \delta\alpha$ **,and**
  ii) either $\mathtt{a}$ is the first symbol of $\mathtt{w}$, or $\mathtt{w}$ is ϵ and $\mathtt{a}$ is $

LR(1) sets of items collection

The basic procedure is same as for LR(0) sets of items collection, if we specify how to compute the lookahead for the new item obtained by closure.

- **If $A \to \alpha \bullet B\beta$, $\mathtt{a}$ is a valid item for a viable prefix γ, then the item $B \to \bullet\eta$, $\mathtt{b}$ is also valid for the same prefix γ. Here $B \to \eta$ is a rule and $\mathtt{b} \in$ FIRST($\beta\ \mathtt{a}$).**

CONSTRUCTING LR(1) SETS OF ITEMS

The algorithm is similar to Algorithm 3.3 and given in the following.

LR(1) PARSER CONSTRUCTION

CANONICAL COLLECTION OF LR(1) SETS OF ITEMS

```
procedure items(G');
    begin
        C := { closure ( S' → •S, $ )};
        repeat
            for  each set of items I in C and each grammar
            symbol X, such that goto(I,X) is not empty
            and is not already in C, do add goto(I,X) to C
        until no more sets of items can be added to C
    end
```

```
function closure(I);
    begin
        repeat
            for each item A → α • Bβ, a
            and each rule B → γ and each b ∈ FIRST(β a)
            such that B → •γ, b is not in I
            do add B → •γ, b to I;
        until no more items can be added to I;
        return I;
    end
```

LR(1) PARSER CONSTRUCTION

LR(1) PARSING TABLE CONSTRUCTION

```
function goto(I,X)
    begin
        goto_items := φ ;
        for  each item A → α • Xβ, a in I,
             such that A → αX • β,a is not in goto_items
             do add A → αX • β,a to goto_items
        goto := closure(goto_items);
        return goto;
    end
```

- **The application of this algorithm to construct a canonical collection of sets of LR(1) items is illustrated in the following.**

  1. **Consider the grammar given in Figure 3.22. The first collection, i.e., state I_0, is given by $closure(S' \rightarrow \bullet S, \$)$.**
  2. **This causes the LR(1) items $S \rightarrow \bullet R, \$$ and $S \rightarrow \bullet L = R, \$$ to be added to I_0.**
  3. **Closure of the item $S \rightarrow \bullet R, \$$ causes the addition of the item $R \rightarrow \bullet L, \$$ whose closure in turn adds the items $L \rightarrow \bullet * R, \$$ and $L \rightarrow \bullet$ id,$ to I_0.**

LR(1) PARSER CONSTRUCTION

LR(1) PARSING TABLE CONSTRUCTION

4. **Closure of the item $S \rightarrow \bullet L = R, \$$ leads to the inclusion of items $L \rightarrow \bullet * R, =$ and $L \rightarrow \bullet \texttt{id}, =$. The lookaheads for these items are given by FIRST(= R \$).**

5. **Since some LR(1) items added by steps 3 and 4 above have the same core, but different lookaheads, we combine the lookaheads to write them in a compact form, $L \rightarrow \bullet * R, \$ =$ and and $L \rightarrow \bullet$ id, \$ =**

6. **Collection $I_0 = \{S' \rightarrow \bullet S, \$$; $S \rightarrow \bullet R, \$$;**
   $S \rightarrow \bullet L = R, \$$; $L \rightarrow \bullet * R, \$ =$; $L \rightarrow \bullet$ id, \$ = ;
   $R \rightarrow \bullet L, \$$ }

7. **The *goto* function on I_0 is defined for the symbols S, L, R, id and *. The resulting states are obtained by taking closures of the kernel items as given below. $I_1 = goto(I_0, S) = closure(S' \rightarrow S\bullet, \$)$**
   $I_2 = goto(I_0, L) = closure(S \rightarrow L\bullet = R, \$; R \rightarrow L\bullet, \$\ )$
   $I_3 = goto(I_0, R) = closure(S \rightarrow R\bullet, \$\ )$
   $I_4 = goto(I_0, *) = closure(L \rightarrow * \bullet R, \$ =\ )$
   $I_5 = goto(I_0,\texttt{id}\ ) = closure(L \rightarrow \texttt{id}\bullet$, **\$ =** $)$
   The rest of the collection can be constructed on the same lines.

The canonical collection $C = \{I_0, I_1, \ldots, I_n\}$ is used to construct the Action and Goto part of the table.

LR(1) PARSER CONSTRUCTION

LR(1) PARSING TABLE CONSTRUCTION

The set I_i and its constituent items define the entries for state i of the Action table as follows

1. **if $A \rightarrow \alpha\bullet$ a β, b $\in I_i$ and $goto(I_i$, a$) = I_j$**
   Action[i,a] = shift j
2. **$A \rightarrow \alpha\bullet$,a $\in I_i$**
   Action[i,a] = reduce by $A \rightarrow \alpha$
3. **$S' \rightarrow S\bullet, \$ \in I_i$**
   Action[i,$] = accept
4. **All remaining entries of state i are marked as error.**
5. **The Goto entries of the state i are**
   $goto(I_i, A) = I_j$ leads to Goto[i,A] = j
   The remaining entries are marked as error.

LR(1) automaton for the grammar of Figure 3.22 is given in Figure 3.24. The LR(1) parsing table is given in Figure 3.25.

LR(1) GRAMMAR AND PARSER

A grammar for which there is a conflict free LR(1) parsing table is called a LR(1) grammar and a parser which uses such a table is known as LR(1) parser.

FIGURE 3.24 : LR(1) AUTOMATON FOR GRAMMAR OF FIGURE 3.22

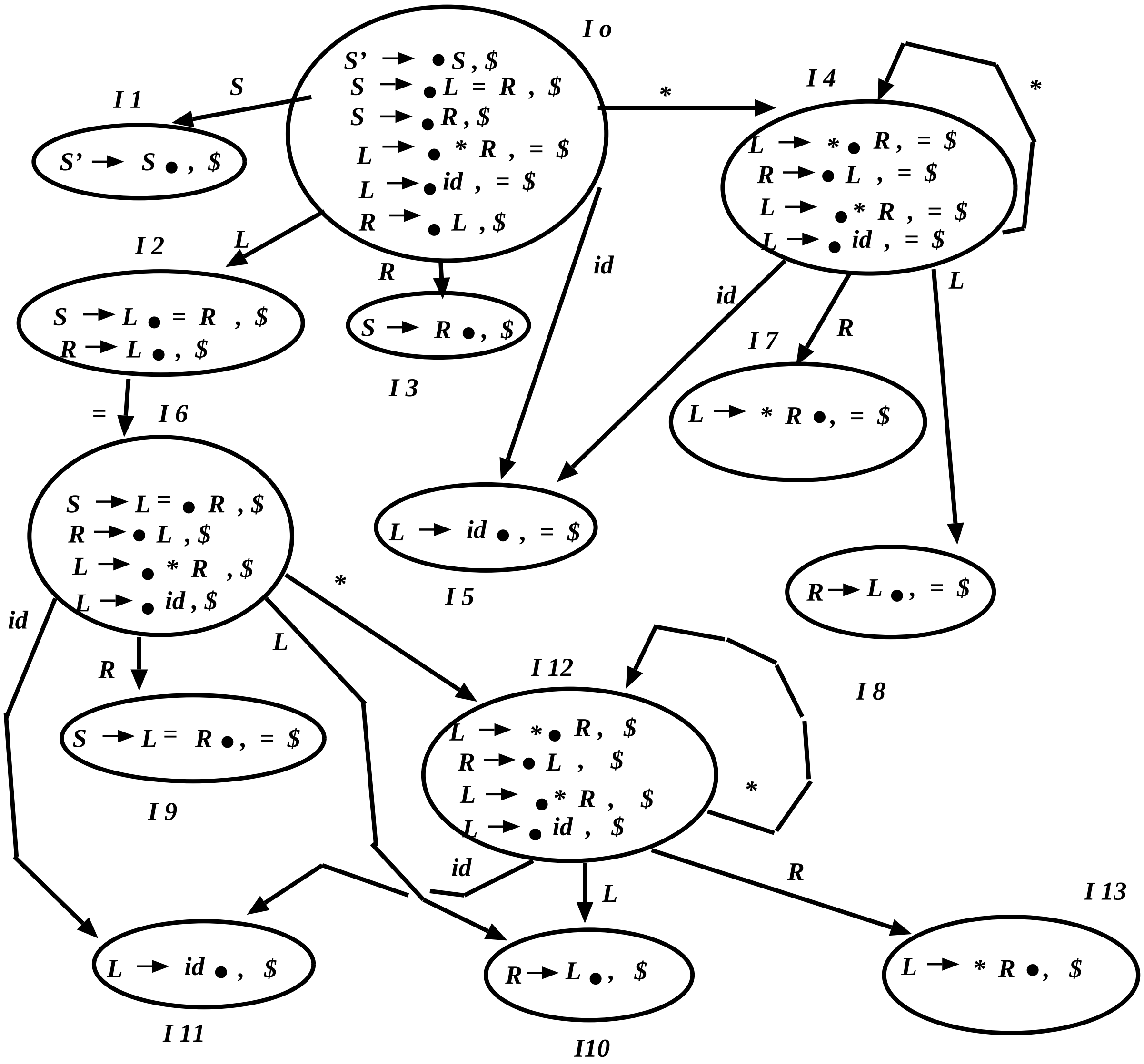


FIGURE 3.25 : LR(1) PARSING TABLE FOR GRAMMAR OF FIGURE 3.22

	Action				Goto		
state	id	*	=	$	*S*	*L*	*R*
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6	r5	-	-	-
3	-	-	-	r2	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r4	r4	-	-	-
6	s11	s12	-	-	-	10	9
7	-	-	r3	r3	-	-	-
8	-	-	r5	r5	-	-	-
9	-	-	-	r1	-	-	-
10	-	-	-	r5	-	-	-
11	-	-	-	r4	-	-	-
12	s11	s12	-	-	-	10	13
13	-	-	-	r3	-	-	-

LALR(1) PARSER CONSTRUCTION

LALR(1) PARSER

- This parser is of intermediate capability as compared to SLR(1) and LR(1). The parser has the size advantage of SLR and lookahead capability like LR.
- In a LR parser, several states may have the same first components (LR(0) items) but differ in the lookaheads associated and hence are represented as distinct states.
- In terms of the first component only, both SLR and LR define the same collection of LR(0) items.
- LALR automaton is created by merging states of LR automaton that have the same core (set of first components). The merge results in the union of the lookahead symbols of the respective items.

LALR(1) PARSING TABLE CONSTRUCTION

1. Construct the collection $C = \{I_0, I_1, \ldots, , I_n\}$ of LR(1) items.
2. For each core among the set of LR(1) items, find all sets having the same core and replace these sets by their union.
3. From the reduced collection, say, $C' = \{J_0, J_1, \ldots, J_m\}$, the Action table is constructed the same way.

LALR(1) PARSER CONSTRUCTION

LALR(1) PARSER

4. **To compute Goto[J,X], let $J = \{I_1 \cup I_2 \cup \ldots \cup I_k\}$. Find K which is the union of all the LR(1) sets that have the same core as $goto(I_1, X)$. Then Goto[J,X] = K.**

To illustrate the construction of a LALR automaton, consider the LR(1) automaton given in Figure 3.24.

1. **This automaton has the following states whose cores are identical. States I_4 and I_{12}, states I_5 and I_{11}, states I_7 and I_{13}, and states I_8 and I_{10}.**

2. **The merger of the states given above results in $I_4 = I_4 \cup I_{12}$, $I_5 = I_5 \cup I_{11}$, $I_7 = I_7 \cup I_{13}$ and $I_8 = I_8 \cup I_{10}$. This happens since in each case the lookahead sets are proper subsets.**

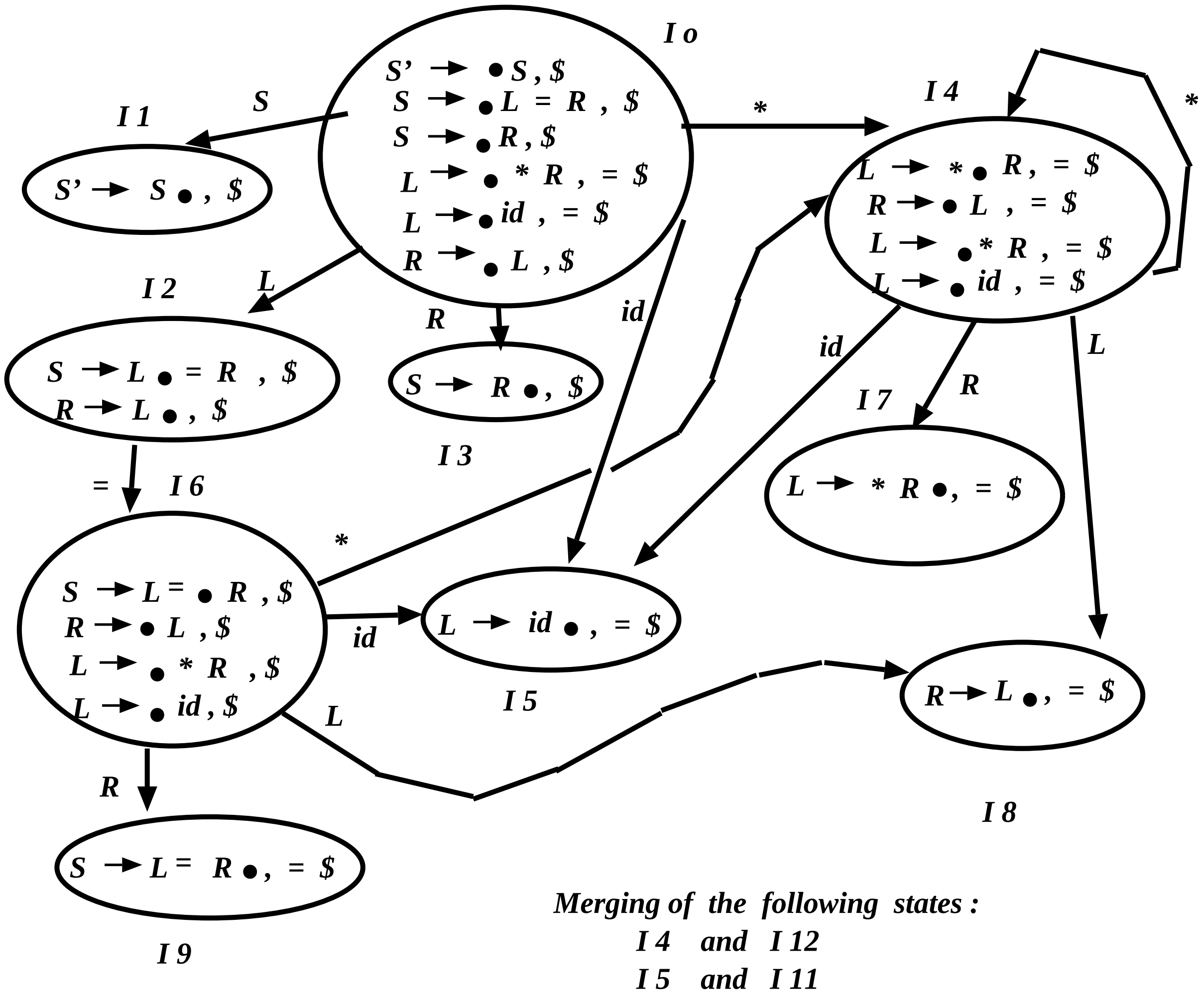
The LALR automaton and the parsing table for the grammar of Figure 3.22 are given in the Figures 3.26 and 3.27 respectively.

LALR(1) GRAMMAR AND PARSER

A grammar for which there is a conflict free LALR(1) parsing table is called a LALR(1) grammar and a parser which uses such a table is known as LALR(1) parser.

LALR(1) PARSER CONSTRUCTION

FIGURE 3.26 : LALR(1) AUTOMATON FOR GRAMMAR OF FIGURE 3.22



Merging of the following states :

I 4 and I 12
I 5 and I 11
I 7 and I 13
I 8 and I 10

FIGURE 3.27 : LALR(1) PARSING TABLE FOR GRAMMAR OF FIGURE 3.22

Grammar :

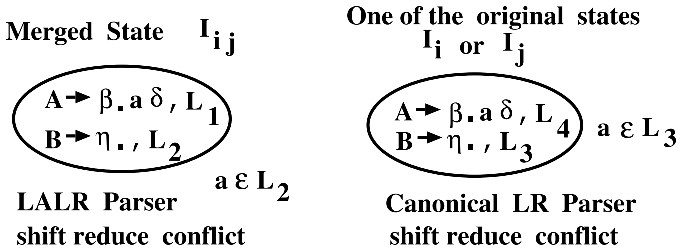
$S' \rightarrow S$
$S \rightarrow L = R \mid R$
$L \rightarrow * R \mid id$
$R \rightarrow L$

state	Action				Goto		
	id	*	=	$	*S*	*L*	*R*
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6	r5	-	-	-
3	-	-	-	r2	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r4	r4	-	-	-
6	s5	s4	-	-	-	8	9
7	-	-	r3	r3	-	-	-
8	-	-	r5	r5	-	-	-
9	-	-	-	r1	-	-	-

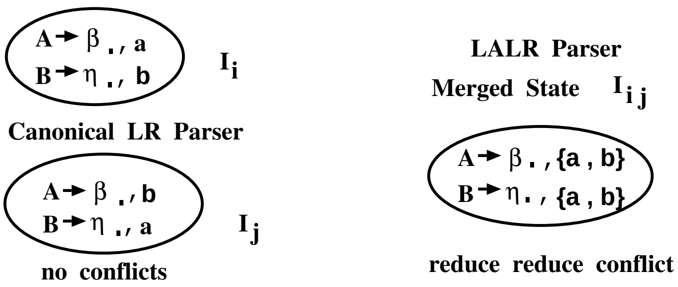
LALR(1) PARSER CONSTRUCTION

RESULTS IN LALR(1) PARSING

- A theorem states that a LALR(1) parsing table is always free of shift reduce conflicts, provided the corresponding LR(1) table is so.
  The key observation towards proving this fact is given in the following.



- However the theorem does not exclude the possibility of a LALR(1) parsing table having reduce reduce conflicts while the corresponding LR(1) table is conflict free as shown below.



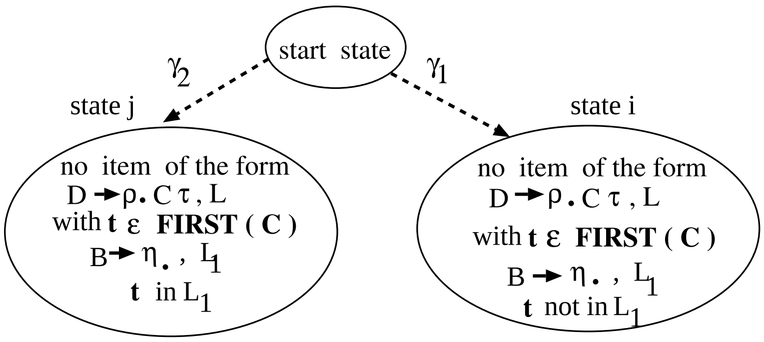
LALR(1) PARSER CONSTRUCTION

RESULTS IN LALR(1) PARSING

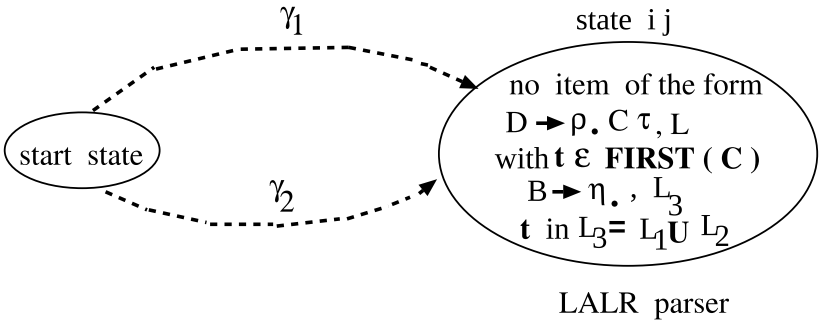
- Another important result relates the behaviour of these two parsers.
- The behaviour of both LALR and LR parsers are identical (same sequence of shifts and reduces) for a sentence of the language.
- However, for an erroneous input, a LALR parser may continue to perform reductions after the LR parser has caught the error.
- An interesting fact is that a LALR parser does not shift any symbol beyond the point that the LR parser declared an error.
- Use the parsers of Figures 3.25 and 3.27 to parse the input `id` = *`id` = $ and note the difference in behaviour.
- The reasons are analyzed in the following. The discussion can be converted into a proof for this result if LALR has only one merged state.
- Let a LR(1) parser be in state i with `t` as the next symbol. This state recognizes a viable prefix γ_1. In the figure given below, state i indicates error on `t` since the first form of items do not permit `t` to be shifted and the other prevents any reduce on `t`.

LALR(1) PARSER CONSTRUCTION

Behaviour of LR and LALR Parsers on Erroneous Inputs



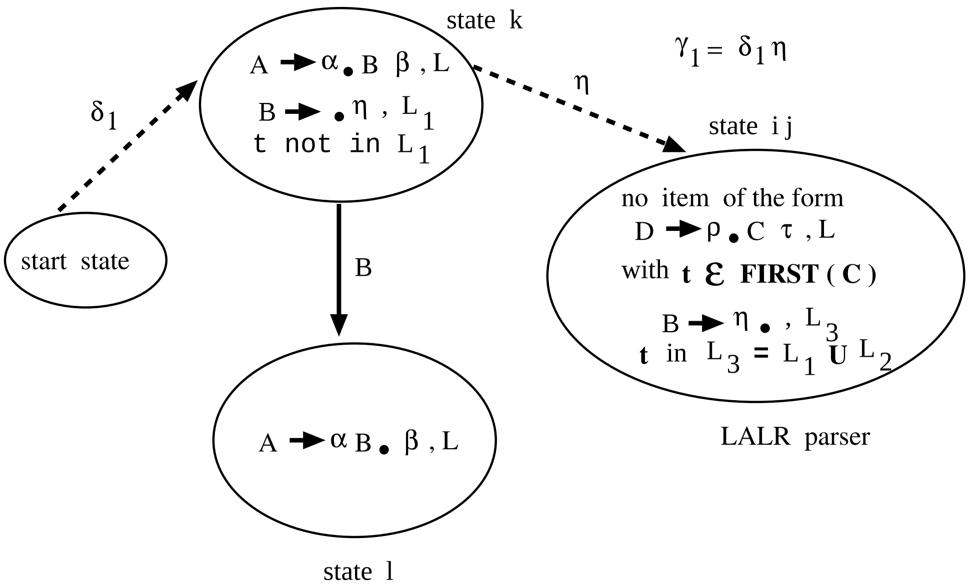
- **State i of the corresponding LALR parser is shown below. This state after recognizing γ_1 permits a reduce on t because of the state j of the LR parser.**



LALR(1) PARSER CONSTRUCTION

Behaviour of LR and LALR Parsers on Erroneous Inputs

- After reducing by $B \rightarrow \eta\bullet$, the LALR parser would be in state l whose relevant contents are shown below. It is assumed that states k and l of both the parsers are identical.



- Presence of the item $A \rightarrow \alpha \bullet B\beta, L$ in state k is justified since $B \rightarrow \bullet\eta, L_1$ is an item which can only be added through closure. Terminal t must not be in FIRST(β) since otherwise t $\in L_1$ which in turn shows that state l does not shift t.

LR PARSERS

COMPARISON OF THE LR PARSERS

1. **SLR parser is the one with the smallest size and easiest to construct. It uses FOLLOW information to guide reductions.**
2. **Canonical LR parser has the largest size among the other parsers of this family. The lookaheads are associated with the items and they make use of the left context available to the parser.**
3. **LR grammar is a larger subclass of context free grammars as compared to that of SLR and LALR grammars.**
4. **LALR parser has the same size as that of a SLR parser but is applicable to a wider class of grammars than SLR grammar.**
5. **LALR is less powerful as compared to LR, however most of the syntactic features of programming languages can be expressed in this grammar.**
6. **LALR parser for an erroneous input behaves differently than a LR parser. Error detection capability is immediate in LR but not so in LALR.**
7. **Both LR and LALR parsers are expensive to construct in terms of time and space. However, efficient methods exist for constructing LALR parsers directly.**

LR PARSERS

AUTOMATIC GENERATION OF THE LR PARSERS

- All the parsers of this family can be generated automatically given a context free grammar as input. The parser generation tool YACC available in UNIX uses such a strategy for generating a LALR parser.
- The basic steps to automate the process for generating a SLR parser from a grammar G are given below.
  1. Augment the given grammar G.
  2. Construct FOLLOW information for all the nonterminals of G.
  3. Use the algorithms for Closure , goto and sets-of-items construction to construct the canonical collection of LR(0) sets of items.
  4. Use the procedure for constructing a parsing table from the canonical collection to create the table. If the table contains conflicts, report them.
  5. Supply a driver routine which uses a stack, an input buffer and the parsing table for actually parsing an input.
- The other parsers can also be generated along similar lines.