6

DESIGN AND IMPLEMENTATION OF PARSER

6.1 Introduction

This chapter describes the design and implementation of the parser. There have been various approaches to the parsing problem. Notable among these studies include the Cocke-Kasami-Younger algorithm, Earley’s algorithm, Valiant’s algorithm, and Tomita’s algorithm (Graham and Harrison [1976] and the references sighted therein, Earley [1970], and Tomita [1986]). Both of Cocke-Kasami-Younger and Earley’s algorithms require an amount of time proportional to the cube of the length of the string being parsed, i.e. $O(n^3)$, with a scope of improvement in time bound in special cases. The Valiant’s algorithm has subcubic time bound, which is $O(n^{2.81})$, to be specific. Tomita’s algorithm has the time complexity to be $O(n^3)$, and is five to ten times faster than Earley’s algorithm. Graham and Harrison establish the upper and lower time bounds for parsing a string of length $n$ to be $O(n^{2.81})$ and $O(n)$ respectively, representing a great gap. All these algorithms are applicable to Context-Free Languages and have their roots in the theory of formal languages. Stabler [1992] maintains that the studies on formal languages are inadequate for natural languages. These algorithms require the grammar in terms of rules. Keeping these observations and the advantages of ‘parametric approach’ over ‘rule-based approach’ to grammar as mentioned in earlier chapters, specifically in Chapter 1, in mind we present a new algorithm for parsing natural languages within GB framework (Singh and Singh [1996b]). The algorithm uses the modified X-bar rule schemata, and the Argument Structure and the parameter settings for the language under consideration as formulated in earlier chapters. The algorithm accepts the grammar in the form of parameter settings (Singh and Singh [1996a]). For any language, the grammar is supplied on-line. The algorithm works for different languages. The parser based on
this algorithm is compared, individually, with the parser used in KBMT system and
the one used in UNITRAN MT system, and is seen to be remarkably different. The
parser is currently tested for three languages namely Punjabi, Hindi, and English.
Other works on parsing within GB framework include Barton [1984], Berwick,
Abney, and Tenny [1991], Berwick and Fong [1990], Dorr [1987a, 1988, 1990b,
1991a, 1993a,b], Fong [1991], Kashket [1991], Merlo [1992], Shaban [1991], Sharp
[1985], and Sigurd and Mats [1991].

Section 6.2 below describes the parsing process. Section 6.3 presents the
algorithm developed for parsing natural languages. Section 6.4 mentions the typical
characteristics of the algorithm. Section 6.5 discusses the experimentation details and
lists out possible enhancements. Section 6.6, finally, concludes the chapter.

6.2 The Parsing Process

The parsing process is depicted in Figure 6.1. The parser takes as input the given
sentence, the Argument Structure, and the 'parameter settings' for the chosen
language. The Argument Structure takes care of the parameter settings of the
Complement parameter associated with X-bar rule schemata. The 'parameter settings'
in the parsing process jointly refer to parameter settings of the Specifier parameter and
the Adjunct parameter associated with the X-bar rule schemata. Since the Argument
Structure varies with respect to a lexical/ non-lexical head and parameter settings vary
with respect to each phrase, the information about the Argument Structure is treated
separately from that about the parameter settings. For lexical heads, the Argument
Structure is stored in the lexicon and for non-lexical heads I and C it is stored as part
of the working memory. For each phrase, the parameter settings are stored as part of
the working memory. As the X-bar rule schemata is language-independent, it is
embedded into the parser itself. The parser produces as output a parse tree stored in
the form of a binary tree which is displayed in the bracketed as well as in the graphical
representation. A phrase is defined immediately once we know the parameter settings
associated with the X-bar rule schemata. At any moment of time, we always keep in
mind the X-bar rule schemata, the Argument Structure, the parameter settings for the parameters, and the Projection Principle.

6.3 The Algorithm

The parsing algorithm consists of the following steps:

(i) Select the language of choice and load its parameter settings.

(ii) Accept the given sentence $S$ of the language under consideration.

(iii) Break $S$ into words (lexical items) using the possible delimiters (punctuation symbols).

(iv) For each word in the sentence, get its Argument Structure from the lexicon. If it is not available there, get it from the user and store it in the lexicon for later use.

(v) Call Parse-sentence to give the parse tree. Parse-sentence takes sentence as input in the form of words obtained in Step (iii), and gives the parse tree as output. The parse tree is perceived in the form of a binary tree.
(vi) Show the parse tree in bracketed as well as in graphical notation.

(vii) Show the remaining part (words) of the unparsed sentence.

(viii) Show appropriate messages in case some error occurs. For example, if a sentence is not having the grammatical subject, the message 'Extended Projection Principle not satisfied' is shown; or if a lexical item is shown to have internal argument, but it is not present in the sentence, the message 'Projection Principle not satisfied' is shown.

*Parse-sentence* at Step (v) is a module which acts as a driver. It calls the *Get-phrase* module.

We apply the **bottom-up** parsing technique which begins by attempting to combine words into larger units and then works “up” to still larger units thus giving phrases and finally the sentence. The sentence is scanned in the direction from **right to left**. We have preferred right to left parsing over the left to right parsing as the latter demands more look-ahead than the former. Consider, for example, the Punjabi sentences “*uh Daaktar hai.*” (“He is a Doctor.”) and “*uh Daaktar baRaa kharaab hai.*” (“He is a very bad Doctor.”). When parsing from left to right, we can not know whether ‘*uh Daaktar*’ is to be treated as NP (the subject), or *uh* as NP and *Daaktar* as the Complement NP of verb *hai* until we reach the third word in the sentences. In the first example, *uh* (“he”) acts as subject NP whereas in the second ‘*uh Daaktar*’ (“that doctor”) acts as the subject NP. Crain and Fodor [1985] also mention about such a limitation of “from left to right” parsing. The parser incorporates the **Wait and See Parsing (WASP)** strategy (Winston [1984]). When a head does not obey either the Adjacency Requirement on Complements or the argument hierarchy imposed by the Argument Structure, the parser cannot declare the sentence to be invalid at that very moment. Rather, it has to “wait” and “see” for the expected constituent (argument) at some other position in the sentence. The parser keeps on waiting till the expected constituent is obtained, or no more words in the sentence remain to be scanned. In the former case, the constituent is adjoined at the appropriate position as discussed in Chapter 5. In the latter case, the sentence is declared to be invalid and appropriate
error message is generated. This strategy is used to handle the scrambling phenomenon found in natural languages.

For each lexical item $X^0$, its Argument Structure is consulted. Using the parameters of X-bar theory and their parameters settings, XP is formed using the definition of XP as defined by X-bar rule schemata. For a given lexical item $X^0$ with known Argument Structure, once the non-head constituents in any of the rules in the X-bar rule schemata are known, the parse (i.e. the syntactic structure) is obtained immediately.

For a given lexical item $X^0$, Get-phrase generates an XP in the case where each non-head constituent in the X-bar rule schemata is head-first. For the current word $X^0$, it recursively calls itself to generate in turn its Complement(s), and then the Adjunct(s) and the Specifier for XP, and finally constructs XP. In addition, it also takes care of the situation where Adjacency Requirement on Complements is not met. In such a case the Complement is adjoined at XP level with a trace at the Complement position as described in Chapter 5.

Parse-sentence goes on accumulating phrases in case the current word $X^0$ expects its Complement(s), Adjunct(s) and Specifier to be head-last, which have already been generated and remembered by Get-phrase. The driver stops as soon as it obtains the Verb Phrase. After this it generates IP and CP by consulting their parametric values.

6.4 Characteristics of the Algorithm

The parsing technique presented in this chapter has the following typical characteristics:
(i) It accepts the grammar in the form of parameter settings for any language under consideration.

(ii) The grammar can be supplied on-line instead of hand-construction of individual grammars perceived in the form of rules for the languages to be attached to the parser. There are a few parameters, each with a limited set of parameter settings.

(iii) The algorithm works cross-linguistically as it is based on the universal X-bar Theory of phrase structure.

(iv) Any language can be attached to the parser by supplying its parameter settings and the Argument Structure.

(v) Assuming each dictionary look-up takes $O(1)$ time, the complexity of the algorithm is $O(n)$, where $n$ is the number of words in the sentence to be parsed. This is possible as a dictionary (lexicon) can be implemented using the hashing technique.

(vi) The algorithm and the parser based on this algorithm differs remarkably from the parser used in KBMT system reported by Nirenburg (Dorr [1993b], Nirenburg [1987], Nirenburg et al. [1992], Slocum [1988]) and the one used in UNITRAN MT system developed by Dorr (Berwick and Fong [1990], Dorr [1990b, 1993a,b]). The single parser used in KBMT system is essentially a ruled-based parser that requires the hand-construction of separate computational grammars for the languages under consideration perceived in the form of thousands of language-specific rules. Recent studies show that rule-based approaches to express the grammar of a language are no longer sufficient (Berwick and Fong [1990], Cook [1988], Chomsky [1986b], and Dorr [1993b]). Though Dorr’s works categorically criticize rule-based approaches, Dorr’s parser uses rules indirectly. Using modified version of Earley’s algorithm by Marcken [1989], it generates context free rules by instantiating the X-bar rule schemata with values $X = N, V, ADJ, P, \text{etc.}$ In contrast, not even a single rule is used in our parser. The X-bar rule schemata is treated as a principle; it is taken as innate. The parser accepts the grammar strictly in the form of parameter settings. The way the Argument Structure is applied to the process of parsing influences significantly the performance of the parser. Unlike the parser used in
Dorr's UNITRAN system which applies Argument Structure later to the underspecified X-bar structures generated by the X-bar component to discard illegal structures, the algorithm presented here applies Argument Structure at once a new lexical item/functional head comes into picture. This simply eliminates overgeneration as found in Dorr's parser. The role of Argument Structure in the parsing process has already been described in Chapter 3. Finally, Dorr's parses do not satisfy the Binary Branching Requirement. Such parses are considered invalid by Duarte [1991] and Kayne [1983]. All syntactic structures generated by our algorithm satisfy this requirement.

6.5 Implementation and Experimentation

The parsing algorithm described above has been implemented in C++ leading to a working parser currently tested for three languages namely Punjabi, Hindi, and English. C++ is one of the most commonly used programming languages. It provides direct implementation of Abstract Data Types (Aho, Hopcroft, and Ullman [1983], Booch [1991], and Khoshafian and Abnous [1990]). The language has the peculiar features of modular design, abstraction, encapsulation, polymorphism, inheritance, portability, reusability, reliability, and readability. These features of the language facilitate the system expandability. The class facility of C++ makes the debugging process easier (Berry [1988], Lippman [1989], and Stroustrup [1992]).

Below we present the implementation details of the parsing algorithm. The algorithm is described in terms of C++ classes. To be brief and for conceptual clarity, the procedural details have been omitted. Only the declarations of class members and their function has been mentioned. In addition, declarations of each of initialization and global variables, and the supporting functions with their definitions have also been provided. A total of three classes have been identified. These classes are the sentence class, the lexicon class, and the parse class. The sentence class has six member functions, namely sentence(), get_sentence(), put_sentence(), get_words(); nw(), and show_word(). The lexicon class contains four member functions, namely open_file(), close_file(), get_info(), and get_and_put_info(). The parse class has a
total of nine member functions, namely get_phrase(), display(), get_tree(), set_tree(),
set_rp(), get_rp(), get_phrase_type(), parse_sentence(), and show_remaining_words().
In addition, the system includes nine supporting functions which have been used
throughout the parser. These supporting functions are
choose_and_load_language_data(), load_punjabi_data(), load_hindi_data(),
load_english_data(), make_tree(), show_or_get_word_info(), depth_of_tree(),
draw_node(), and draw_tree().

/* Defining constants. */
#define MAX_LENGTH 80  //Sets the maximum length of a sentence.
#define MAX_WORDS 20   //Sets the maximum number of words
                      //a sentence can have.
#define MAX_LC_LENGTH 8  //Sets the maximum length of a lexical
                      //category/ the corresponding phrase.
#define MAX_WORD_LENGTH 20 //Sets the maximum length of a word.
#define MAX_ARG_LENGTH 9  //Sets the maximum length of an
                      //argument/ non-head constituent in X-
                      //bar Rule Schemata.
#define MAX_NO_OF_ADJNS 5 //Sets the maximum number of Adjuncts
                      //a phrase can have.
#define MAX_NO_OF_SPECFRS 5 //Sets the maximum number of
                         //Specifiers a phrase can have.
#define NO_OF_LC 12   //Sets the maximum number of lexical
                        //categories/ the corresponding phrases.

/* The end of constant definitions. */

206
/* Type declarations. */

typedef struct treenode  //Declaration for a binary/ parse tree.
{
    char        nodeinfo[MAX_WORD_LENGTH];  //Node information.
    struct treenode *left;  //The left pointer.
    struct treenode *right;  //The right pointer.
};

typedef struct ENTRY  //Declaration for the lexical entry, i.e. the
{  //Argument Structure of a lexical item.
    char    w[MAX_WORD_LENGTH];  //The lexical item, i.e. the word.
    char    lc[MAX_LC_LENGTH];  //The lexical category of the word.
    int    no_of_args;  //The number of arguments.
    char    arg[2][MAX_ARG_LENGTH];  //The type of each argument.
};

typedef struct partial_tree_array  //Declaration for holding partial trees/
{  //parses.
    treenode       *partial_tree;
    char    lc[MAX_LC_LENGTH];  //The type of the partial parse.
};
struct adj_spec //Declaration for providing the parameter
    //settings of Specifier parameter and the
    //Adjunct parameter.
{
    char lc[MAX_LC_LENGTH]; //The type of the phrase.
    int no_of_adjuncts; //The number of Adjuncts.
    char adj_values[MAX_NO_OF_ADJNS][MAX_ARG_LENGTH];
        //The type of each Adjunct.
    int no_of_specifiers; //The number of Specifiers.
    char spec_values[MAX_NO_OF_SPECFRS][MAX_ARG_LENGTH];
        //The type of each Specifier.
}array_of_adj_spec[NO_OF_LC]; //To specify the parameter settings for
    //various phrases.

/* The end of type declarations. */

/* Class definitions. */

class sentence //The sentence class.
{
    char s[MAX_LENGTH]; //The sentence.
    char delimiters[10]; //The delimiters/ punctuation symbols.
    int n; //The number of words in the sentence.
    char *words[MAX_WORDS]; //The words in the sentence.

public:

    sentence() {n=0; strcpy(delimiters, ",."陌生人;)}

        //Initializes the number of words and the
//delimiters.

void get_sentence() {cout << "Input Sentence: "; gets(s);}
//Reads a sentence.

void put_sentence() {cout << s;} //Writes the sentence.

void get_words(); //Breaks the sentence into words.

int nw() {return n;}
//Returns number of words in the
//sentence.

char *show_word(int i) {return words[i];}
//Returns ith word of the sentence.

};

class lexicon //The lexicon class.
{

FILE *fp; //The file pointer to the lexicon.

public:

int open_file(char *file_name); //Opens the file.

void close_file() {fclose(fp);} //Closes the file.

int get_info(char *, ENTRY *); //To obtain the Argument Structure of a
//lexical item from the lexicon.

void get_and_put_info(char *); //Reads the Argument Structure of a
//lexical item from the user and stores it
//in the lexicon.

};
class parse  //The parse class.
{
    treenode *prs;  //Points to the parse tree.
    char final_phrase[8];  //Type of the final phrase
    int rp;  //Type of the index of the current word.

public:
    treenode *get_phrase(int, char *);  //Returns the phrase starting at the given
    //index of a word in the sentence along
    //with its type.
    void display(treenode *);  //Displays a parse in the bracketed
    //notation.
    treenode *get_tree();  //Returns a pointer to the parse tree.
    void set_tree(treenode *p) {prs = p;}  //Initializes a parse to be the parse tree.
    void set_rp(int i) {rp = i;}  //Sets the index of the word in the
    //sentence from where the scanning is to
    //start.
    int get_rp() {return rp;}  //Returns the index of the word in the
    //sentence from where the scanning is to
    //start.
    char *get_phrase_type() {return final_phrase;}  //Gives the type of final phrase.
    void parse_sentence(char );  //Parses the sentence in a given language.
    void show_remaining_words();  //Displays the remaining part (words) of
    //the unparsed sentence.
};
int depth=0, deflection, ht_partition, size;

void draw_tree(treenode *);

/* The end of declarations used for graphical display of the parse. */

/* Other global declarations. */

treenode *make_tree(char *info, treenode *l, treenode *r);

sentence sen; //Of type sentence class.
lexicon lxn; //Of type lexicon class.

/* The end of other global declarations. */

/* Supporting Functions */

char choose_and_load_language_data(char *file_name)

//Prompts for and returns the language of interest, and loads the grammar
//((parameter settings) of the chosen language using the functions
.sourceforge punjabi_data(), load_hindi_data(), or load_english_data().

void load_punjabi_data()

  //Fixes for each phrase the parameter settings for Punjabi language.
{
  strcpy(array_of_adj_spec[0].lc, "VP");
  array_of_adj_spec[0].no_of_adjuncts = 4;
  strcpy(array_of_adj_spec[0].adj_values[0], "ADVP");
  strcpy(array_of_adj_spec[0].adj_values[1], "PP");
strcpy(array_of_adj_spec[0].adj_values[2], "NEGP");
strcpy(array_of_adj_spec[0].adj_values[3], "EMPP");
array_of_adj_spec[0].no_of_specifiers = 0;

strcpy(array_of_adj_spec[1].lc, "NP");
array_of_adj_spec[1].no_of_adjuncts = 3;
strcpy(array_of_adj_spec[1].adj_values[0], "ADJP");
strcpy(array_of_adj_spec[1].adj_values[1], "PP");
strcpy(array_of_adj_spec[1].adj_values[2], "IP");
array_of_adj_spec[1].no_of_specifiers = 2;
strcpy(array_of_adj_spec[1].spec_values[0], "CASEP");
strcpy(array_of_adj_spec[1].spec_values[1], "DETP");

strcpy(array_of_adj_spec[2].lc, "CASEP");
array_of_adj_spec[2].no_of_adjuncts = 0;
array_of_adj_spec[2].no_of_specifiers = 0;

strcpy(array_of_adj_spec[3].lc, "ADJP");
array_of_adj_spec[3].no_of_adjuncts = 2;
strcpy(array_of_adj_spec[3].adj_values[0], "PP");
strcpy(array_of_adj_spec[3].adj_values[1], "ADVP");
array_of_adj_spec[3].no_of_specifiers = 3;
strcpy(array_of_adj_spec[3].spec_values[0], "QP");
strcpy(array_of_adj_spec[3].spec_values[1], "PP");
strcpy(array_of_adj_spec[3].spec_values[2], "NP");
strcpy(array_of_adj_spec[4].lc, "DETP");
array_of_adj_spec[4].no_of_adjuncts = 0;
array_of_adj_spec[4].no_of_specifiers = 1;
strcpy(array_of_adj_spec[4].spec_values[0], "ADVP");

strcpy(array_of_adj_spec[5].lc, "QP");
array_of_adj_spec[5].no_of_adjuncts = 0;
array_of_adj_spec[5].no_of_specifiers = 1;
strcpy(array_of_adj_spec[5].spec_values[0], "ADVP");

strcpy(array_of_adj_spec[6].lc, "CONJP");
array_of_adj_spec[6].no_of_adjuncts = 0;
array_of_adj_spec[6].no_of_specifiers = 1;
strcpy(array_of_adj_spec[6].spec_values[0], "NP");

strcpy(array_of_adj_spec[7].lc, "ADVP");
array_of_adj_spec[7].no_of_adjuncts = 0;
array_of_adj_spec[7].no_of_specifiers = 0;

strcpy(array_of_adj_spec[8].lc, "PP");
array_of_adj_spec[8].no_of_adjuncts = 0;
array_of_adj_spec[8].no_of_specifiers = 2;
strcpy(array_of_adj_spec[8].spec_values[0], "DETP");
strcpy(array_of_adj_spec[8].spec_values[1], "ADVP");
strcpy(array_of_adj_spec[9].lc, "NEGP");
array_of_adj_spec[9].no_of_adjuncts = 0;
array_of_adj_spec[9].no_of_specifiers = 0;

strcpy(array_of_adj_spec[10].lc, "EMPP");
array_of_adj_spec[10].no_of_adjuncts = 0;
array_of_adj_spec[10].no_of_specifiers = 0;

strcpy(array_of_adj_spec[11].adj_values[0], "ADVP");
strcpy(array_of_adj_spec[11].spec_values[0], "NP");
strcpy(array_of_adj_spec[11].spec_values[1], "CASEIP");
strcpy(array_of_adj_spec[11].spec_values[2], "CONJP");

strcpy(array_of_adj_spec[12].lc, "CP");
array_of_adj_spec[12].no_of_adjuncts = 0;
array_of_adj_spec[12].no_of_specifiers = 0;
}

void load_hindi_data()

    //Fixes for each phrase the parameter settings for Hindi language.
void load_english_data()

    //Fixes for each phrase the parameter settings for English language.

treenode *make_tree(char *info, treenode *l, treenode *r);

    //Builds a tree with info as nodeinfo, l tree as the left child and r tree as the
    //right child.

void show_or_get_word_info(char *file_name)

    //Gives the Argument Structure of a lexical item from the lexicon, if present
    //there. Otherwise, displays the appropriate message and reads it from the user
    //and stores it in the lexicon.

void depth_of_tree(treenode *tree, int alevel)

    //This function finds the depth of the tree and initializes the global variable
depth //with this value.

void draw_node(treenode *tree, int ax, int y, int alevel)

    //This function draws a node at point (x, y).

void draw_tree(treenode *tree)

    //Draws a tree in graphical representation. Calls other two functions namely
    //depth_of_tree() and draw_node().

/* The end of Supporting Functions */

/* The main function */

void main()
char file_name[30];

parse parse_tree;

char choose_and_load_language_data(char *);

void show_or_get_word_info(char *);

char language = choose_and_load_language_data(file_name);

sen.get_sentence();

sen.get_words();

show_or_get_word_info(file_name);

parse_tree.parse_sentence(language);

parse_tree.display(parse_tree.get_tree());

cout << "\nPress any key to continue."

getchar();

int gdriver=DETECT, gmode;

initgraph(&gdriver, &gmode, "c:\tcp\bgi");

draw_tree(parse_tree.get_tree());

cout << "\nPress any key to continue."

gchar();

closegraph();

parse_tree.show_remaining_words();

lxn.close_file();

} /* The end of the main function */
Each module (member functions of each class plus the supporting functions) is developed and run tested independently of one another. The inputs and outputs of each module were examined carefully. The complete system has been inspected for a variety of sentences. The parses obtained from the parser are found to be compatible with those found in standard works on GB Theory. The system shows encouraging results. Some sample example inputs/outputs are shown below.

**Example 1**

Input: uh kall bhii nahiin aaegaa.

he tomorrow also not come-fut.ms

(‘He will not come tomorrow also.’)

Output: The parse tree in bracketed notation looks as follows.

\[
(I'(N"(N(N(uh))))(I'(V"(V'(ADV'(ADV'(ADV(kall)))))(V'(EMP'(EMP'(EMP'(bhii))))(V'(NEG'(NEG'(NEG'(nahiin))))(V'(V(aaegaa))))))) (l(e)))
\]

The output in graphical notation looks as in Figure 6.2.

![Figure 6.2](image-url) The output parse for Punjabi sentence “uh kall bhii nahiin aaegaa.”
Example 2

Input: us kuRii ne Ram nuun kitaab dittii.

that girl ERG Ram DAT book give-pst.fs

('That girl gave Ram a book.')

Output: The parse tree in bracketed notation looks as follows.

(I"(CASE"(CASE'(N"(DET"(DET'(DET(us))))(N'(N( kuRii))))(CASE(ne)))))

(I'(V"(CASE"(CASE'(N"(N(Ram))))(CASE(nuun))))

(V'(N"(N'(N(kitaab))))(V(dittii))))(1(e))))

The output in graphical notation looks as in Figure 6.3.

Figure 6.3 The output parse for Punjabi sentence “us kuRii ne Ram nuun kitaab dittii.”
Example 3

Input: Ravinder ne caawal nahiin khaadhe.

Ravinder NOM rice not ate

(‘Ravinder did not eat the rice.’)

Output: The parse tree in bracketed notation looks as under.

\[
(I'(CASE"(CASE'N'(N'(N(Ravinder)))))(CASE(ne))))(I'(V'"(N"(N'(N
(caawal))))(V'"(V'(NEG"(NEG'(NEG(nahiin)))))(V'(NP(t))(V(khaadhe)))))))
\]

The output in graphical notation looks as in Figure 6.4.

![Figure 6.4](image-url)

Figure 6.4 The output parse for Punjabi sentence “Ravinder ne caawal nahiin khaadhe.”

This is an example where the adjacency requirement on Complements is not satisfied. The adverb nahiin (‘not’) intersperses between the head verb (V) khaadhe (‘to eat’) and its Complement Noun Phrase (NP) caawal (‘rice’). The Complement NP (caawal) is adjoined at V’ level with a trace t at the Complement position.
Example 4

Input: Mohan sleeps daily in the evening.

Output: The parse tree in bracketed notation looks as follows.

\[(I''(N''(N''(N'(N(Mohan))))))(I'(I(e)))(V''(V''(V''(V'(V(sleeps))))))(ADV''(ADV'(ADV (daily))))))(P''(P'(P(in)))(N''(DET''(DET'(DET(the)))(N'(N(evening)))))))\]

The output in graphical notation looks as in Figure 6.5.

![Figure 6.5 Output parse for “Mohan sleeps daily in the evening.”](image)

Example 5

Input: Yesterday, Mohan showed Mary a movie.

Output: The parse tree in bracketed notation looks as follows.
The output in graphical notation looks as in Figure 6.6.

Figure 6.6 Output parse for "Yesterday, Mohan showed Mary a movie.”

6.6 Conclusions and Further Enhancements

This chapter introduced a new algorithm for parsing natural languages. The algorithm is based on X-Bar Theory, the Argument Structure, and the Projection Principle. The algorithm accepts the grammar in the form of parameter settings. For any language, the grammar is supplied on-line. This eliminates the need for hand-construction of grammars perceived in the form of thousands of language-specific rules for each phrase in a language. The algorithm works cross-linguistically. The time complexity of the algorithm is O(n). The algorithm is implemented in C++ leading to a working parser currently tested for three languages namely Punjabi, Hindi, and English. We parse from right to left, applying the bottom-up parsing technique, and simultaneously incorporating the Wait and See Parsing strategy. The parser takes as input the
sentence, the parameter settings, and the Argument Structure for the language under consideration, and generates as output the parse tree in the form of a binary tree which is displayed in bracketed as well as in graphical notation. The parser based on this algorithm is seen to be remarkably different from the parser used in the KBMT system, and the one used in UNITRAN system. Each syntactic structure generated by the parser satisfies the Binary Branching Requirement in case of Double Object Constructs. The Argument Structure is applied at once as soon as a lexical item comes into picture. This checks the overgeneration and leads to better performance of the parser.

The work assumes the existence of a morphological analyzer for the languages under consideration. In case a lexical item has multiple entries for its Argument Structure, the parser works correctly if the Argument Structure for the lexical item is supplied off-line. To handle such cases on-line, we need to allow the lexicon store the multiple entries for the same lexical item and select the appropriate entry when required. The algorithm does not incorporate the agreement features into the phrases. Punjabi and Hindi do not have the Wh-movement while they both have the NP-movement. On the other hand, English has Wh- as well as NP-movement. The parser handles movement partially. Though the parser handles successfully the declarative as well as interrogative sentences in Punjabi and Hindi, it, in its present status, does not handle interrogative sentences in English. The concept of NP-movement has also not been implemented yet. These features can be handled by considering another parameter namely the 'movement' parameter, specifically the Wh-movement parameter as suggested by Webelhuth [1995a: 28]. English does not allow scrambling, while both Punjabi and Hindi have the phenomenon of scrambling. The parser does incorporate, though partially, the concept of scrambling. The incorporation of these features is underway. The Argument Structure for various lexical items is stored as a sequential file. We intend to modify it to store it as a direct file using a hashing technique to improve the search time.