New Generation Computing, 1 (1983) 145-158
OHMSHA, LTD. and Springer-Verlag

NEW GENERATION COMPUTING

BUP : A Bottom-Up Parser Embedded in Prolog

Yuji MATSUMOTO and Hozumi TANAKA†
Electrotechnical Laboratory, Sakura-mura, Niihari-gun, Ibaraki 305, Japan
Hideki HIRAKAWA, Hideo MIYOSHI, and Hideki YASUKAWA
ICOT Research Center,
Institute for New Generation Computer Technology,
Mita Kokusai Bldg. 21F, 4-28 Mita 1-chome, Minato-ku, Tokyo 108, Japan

Abstract A parser based on logic programming language (DCG) has very useful features ; perspicuity, power, generality and so on. However, it does have some drawbacks in which it cannot deal with CFG with left recursive rules, for example. To overcome these drawbacks, a Bottom-Up parser embedded in Prolog (BUP) has been developed. In BUP, CFG rules are translated into Prolog clauses which work as a bottom-up left corner parser with top-down expectation. BUP is augmented by introducing a "link" relation to reduce the size of a search space. Furthermore, BUP can be revised to maintain partial parsing results to avoid computational duplication. A BUP translator and a BUP tracer which support the development of grammar rules are described.

§1 Introduction

For high quality language analysis, it is necessary to have a flexible and efficient parsing system as well as a powerful descriptive language. Prolog is a language based on first order predicate logic and has very useful features, especially for symbolic manipulation. DEC-10 Prolog[1] has an embedded parsing system called definite clause grammars (DCG).[2] In DCG, each context-free grammar rule is represented as a clause of a Prolog program, and each grammar category in a rule is treated as a predicate in the clause. Therefore, a context-free grammar is translated into a Prolog program in a one-to-one manner. Since predicates of a Prolog program have arguments, context-sensitive information can be handled easily. Furthermore, arbitrary Prolog predicates can be inserted in a grammar rule. These facilities make it easy to combine the parser with such auxiliary routines as semantic checking or structure building. Despite such advantages, DCG has some drawbacks. It cannot deal with a grammar that

† Present address : Department of Computer Science, Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro-ku, Tokyo 152, Japan

includes left recursive rules, for DCG is a top-down parser. Another is that grammar rules and the dictionary are not treated separately. In a large natural language analysis system, the dictionary will be much bigger than the grammar. If a large vocabulary is needed in a system, the dictionary should be handled independently.

To overcome these weak points of DCG, the parsing system BUP embedded in Prolog, which employs a bottom-up parsing mechanism, has been implemented.[3] Each grammar rule corresponds to a Prolog clause also in BUP. It can deal with any cycle-free grammar with no e-productions. The dictionary can be handled independently of grammar rules because of the bottom-up parsing method. Moreover, BUP can easily and elegantly be reinforced with the facilities described in Section 3. This is not so easy for DCG, but will be very complicated.

Section 2 explains the basic concept of BUP. First, the basic formalism and the parsing behavior are explained using a pure CFG. Then the description of the augmentation of CFG and the linking relation are introduced.

Section 3 shows some improvements which give BUP the ability to handle morphological analysis or make this system more efficient.

Section 4 explains the utilities for grammar development, such as a BUP translator and a BUP tracer implemented in Prolog.

§2 The BUP System

2.1 The Principle of BUP

In BUP, grammar rules must be context-free. The context-sensitivity can be expressed by putting arguments in grammar categories or inserting some Prolog programs into clauses. It is assumed that the given context-free grammar is cycle-free with no e-productions.

Context-free grammar rules can be expressed in either of the following forms.

(1) $C \rightarrow C1, C2, \ldots, Cn \ (n >= 1)$

(2) $C \rightarrow a$

Upper and lower case letters stand for nonterminal and terminal symbols respectively. To put grammar and the dictionary separately, the authors here limit to a case in which terminal symbols appear only in type (2) rules. As shown in the following section, terminal symbols can also be put in type (1) rules with slight modification. The above rules are transformed into the following Prolog clauses. According to DEC-10 Prolog syntax, lower and upper cases stand for constants and variables respectively. Grammar categories are expressed by the lower case letters.

```
(1')  c1(G,X1,X) :-
          goal(c2,X1,X2), ... ,goal(cn,Xn-1,Xn), c(G,Xn,X).
(2')  dict(c,[a|X],X).
```

Alternatively, they can be expressed in DCG notation, as follows:

(1″) c1(G)➞goal(c2), ... ,goal(cn), c(G).

(2″) dict(c)➞[a].

The predicate 'goal' in the above clause is defined as follows:

(3) goal(G,X,Z) :-
dict(C,X,Y), P=.. [C,G,Y,Z] , call(P).

'=..' is the built-in operator and P becomes a literal whose head is C and whose argument list consists of G, Y, and Z. Therefore, P becomes the form of C(G,Y,Z) when call(P) is executed, provided that C has been instantiated.

The last set of clauses to be added to the above clauses are as follows. They are the terminal conditions of the call of nonterminal symbols, illustrated later. They are referred to as "terminate clauses".

(4) c(c,X,X). (For every nonterminal symbol C)

The BUP system mainly consists of three parts. They are the goal part, the rule part and the dictionary part. Although the dictionary part might be the largest one, it is called only from the goal part. The parsing algorithm of BUP can be expressed by describing the behavior of the goal part and the rule part. The goal part corresponds to predicate 'goal', and the rule part to the set of clauses transformed from the grammar rules. The second and the third variables of each predicate work as d-lists (the lists which represent a string by their difference) to represent a substring of the given sentence.

(Goal part)

This part receives a grammar category and a string to be parsed, and tries to find a prefix string of the given string which belongs to the category. This category is considered to be a goal.

In Practice, it consults the dictionary to get the category to which the first word of the string belongs, and calls the rule part with the category, the goal and the rest of the original string.

(Rule part)

This part receives a grammar category, the final goal and a string to be parsed. The role of this part is to find the prefix string of the given string which satisfies the goal, on the assumption that a string of the given category has already been found. The practical procedure is:

(a) If the given category is equivalent to the goal, then this call terminates successfully. Otherwise, go to (b).

(b) Pick up a grammar rule that has the given category as the first element of its right hand side, then call the goal part for each element of the rest of the right hand side with its own category name.

(c) If all calls of the goal part in (b) succeed, then call the rule part, passing the category on the left hand side of the CFG rule, the original goal and the rest of the given string.

The nondeterminism of a grammar rule application is handled by a Prolog's backtracking mechanism. There are two backtracking possibilities: consulting the dictionary in the goal part and picking up a grammar rule in the rule part.

The parsing algorithm employed by BUP is a left corner bottom-up method, while that of DCG is a top down method. Therefore, BUP can deal with all left recursive rules except cycle rules.

To explain how the above set of Prolog clauses work as a bottom-up parser, the following example is used. Suppose that the given grammar rules are:

(1) S → NP VP
(2) NP → john
(3) VP → walks

then the BUP clauses are as follows:

```
(1')  np(G,X,Z) :-goal(vp,X,Y),s(G,Y,Z).
(2')  dict(np,[john|X],X).
(3')  dict(vp,[walks|X],X).
(4)   goal(G,X,Z) :-
                dict(C,X,Y),
                P=..[C,G,Y,Z],
                call(P).
(5)   np(np,X,X).
(6)   vp(vp,X,X).
(7)   s(s,X,X).
```

(1') to (3') correspond to (1) to (3) respectively; (4) is a definition of the predicate 'goal' and (5) to (7) are terminate clauses.

If the given sentence is "john walks", then BUP is triggered by the following call.

```
? - goal(s,[john, walks],[]).
```

Figure 1 shows the execution flow of the above call. In this figure, "$\Longrightarrow$" indicates the new body created by the unification of a call, "=" means that both sides of this symbol are equivalent, and "$\leftrightarrow$" shows that the both literals indicated by this symbol are resolved successfully.

2.2 Arguments in Nonterminal Symbols and Extra Predicates

Nonterminal symbols in a BUP clause may have arguments to convey some information. A nonterminal symbol in a DCG clause can have any number of arguments, and the nonterminal symbol with n arguments is translated into a predicate of $n+2$ arguments. A BUP nonterminal in general may have any number of arguments. However, in this system the number has been restricted since it is done by the sole predicate 'goal' which searches for a substring belonging to a certain grammar category. For this reason, the arguments in the nonterminal symbol are packed into a list of arguments.

The BUP clauses (1') and (2') in 2.1 now become:

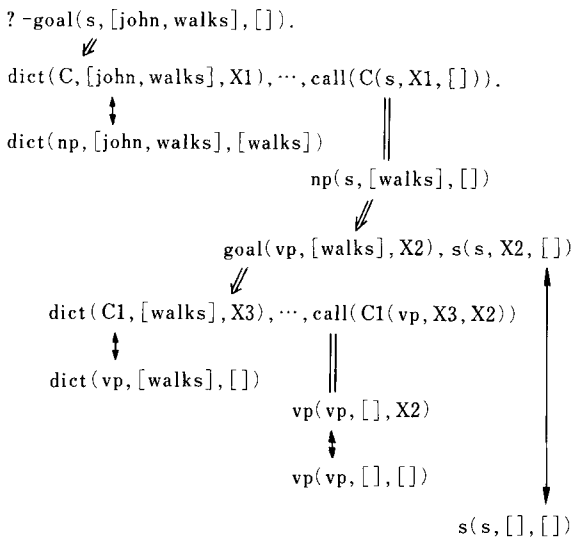


Fig. 1 Execution flow of BUP

(1′) c1(G,AL1,A,X1,X) :-
goal(c2,AL2,X1,X2), ... ,
goal(cn,ALn,Xn−1,Xn),
c(G,AL,A,Xn,X).

(2′) dict(c, AL, [a|X], X).

In the above BUP clauses, each of AL1, ... ,ALn is the list of arguments included in the corresponding nonterminal symbol. Variable 'A' in (1′) works as the conveyer of the information and is instantiated when the call of (1′) succeeds.

Accordingly, goal clause (3) and terminate clauses (4) are now translated as follows:

(3) goal(G,A,X,Z) :-
dict(C,Al,X,Y),
P=.. [C,G,Al,A,Y,X], call(P).

(4) c(c,A,A,X,X). (For every nonterminal symbol c)

Terminate clauses guarantee to pass the information from the second variable to the third variable, which is finally received by the variable 'A' in the goal clause or the variable 'A' in a clause like (1′).

For example, the following grammar rule:

sentence(s(NP,VP))→np(NP,N), vp(VP,N).

is translated into the BUP clause:

np(G, [NP,N], A)→goal(vp, [VP,N]), sentence(G,[s(NP,VP)],A).

When the goal G is 'sentence', the call of this clause will succeed with the terminate clause:

```
sentence(sentence,A,A,X,X),
```

and variable 'A' in the predicate 'np' will be instantiated to '[s(NP, VP)]' after the successful execution of this terminate clause.

2.3 Linking Relation

For eliminating unnecessary work in the parsing process and for making the parser more efficient, BUP can be reinforced with the top-down expectation called 'oracle' in Pratt's algorithm.[4~6)]

The call of a nonterminal symbol in BUP means that a string belonging to the corresponding grammar category has been found. The next task is to select a CFG rule whose leftmost element in the right hand side is the same as that category and to make a call of predicate 'goal' for each of the remaining grammar categories on the right hand side. When all of these calls succeed, a string of the category on the left side has been found. These jobs are done by BUP clauses. When a BUP clause is called, it receives the goal G. If a rule whose category on the lefthand side has no possibility of becoming the left most son of the received goal in parse trees, it should not be selected. This possibility can be known by computing the linking relation between the grammar categories beforehand.

Let the "link" relation hold between the two categories A and B when there is a grammar rule whose form is "B→ A, ...". Assume that the "link" is reflexive and transitive relation. This relation can be computed when the grammar rules are read in BUP. The only other modification is to transform a clause like (1') into (i) and 'goal' into (ii).

```
(i)  c1(G,AL1,A,X1,X) :-
                link(c,G),
                goal(c2,AL2,X1,X2), ... ,goal(cn,ALn,Xn-1,Xn),
                c(G,AL,A,Xn,X).
(ii) goal(G,A,X,Z):-
                dict(C,AL1,X,Y),
                link(C,G),
                P=..[C,G,AL1,A,Y,Z],
                call(P).
```

This 'link' relation works as the local top-down expectation.

§3 Refinements of the BUP System

3.1 Dictionary Looking-up

Indo-European languages and Japanese are inflectional languages. That is, verbs, nouns, and so on in these languages are inflected in many situations. To analyze such languages, parsing systems must have a morphological analysis facility. In BUP, looking-up the dictionary is performed by the predicate 'dict', which is called only in predicate 'goal'. When a 'dict' call fails, a parser must

detect the possibility of inflection. If a procedure 'morph' is constructed to perform the morphological analysis, it is sufficient to modify the predicate 'goal' as follows:

```
goal(G,A,X,Z) :-
    dictionary(C,Al,X,Y), link(C,G),
    P=.. [C,G,Al,A,Y,Z], call(P).
dictionary(C,Al,X,Y):-
    dict(C,Al,X,Y) ; morph(C,Al,X,Y).
```

In the above definition, the predicate 'dictionary' is called many times, and 'dict' is consulted or a morphological analysis performed each time. When the dictionary becomes larger and is put in secondary storage, it is not desirable to do such a time consuming job repeatedly. In order to avoid useless repetition, the predicate 'dictionary' was rewritten to perform all of these tasks for a word when the need to look-up the word occurred.

```
dictionary(C,Al,X,Y):-
    wf_dict(_,_,X,_),!,wf_dict(C,Al,X,Y).
dictionary(C,Al,X,Y):-
    (dict(C,Al,X,Y); morph(C,Al,X,Y)),
    create_dlist(X,Y,U,V),
    assertz(wf_dict(C,Al,U,V)),fail.
dictionary(C,Al,X,Y):-
    wf_dict(C,Al,X,Y).
```

The second definition of the 'dictionary' is the same as the original one, except that all possible dictionary entities and morphological analyses for the given word are performed at this point and are asserted with name 'wf_dict', which stands for a well formed dictionary entity. Predicate 'create_dlist' changes d-lists X and Y to U and V, where V is a variable and is equal to the last element of U. This is because it is enough to assert a well formed dictionary entity with only the string that X and Y represents. The third definition is then called to use the asserted 'wf_dict'. The first definition indicates that once 'dictionary' and 'morph' are executed for a word, it is sufficient to use the only information already asserted.

3.2 Maintaining Success and Failing Goals

The BUP system utilizes the backtracking facility of Prolog implementation to search for all of the possible parsing trees. Although the backtracking facility makes the description of BUP quite simple, it causes inefficiency in the parsing process. The situation is just the same with DCG. When a Prolog program backtracks, the corresponding information is forgotten. This happens in many cases when BUP tries to perform the same job it has already done, that is, BUP frequently searches for the same goal in the same position of the same sentence.

Since the sole predicate 'goal' in BUP searches for a string belonging to

the specified grammar category, improvement can only be achieved by modifying the predicate 'goal' so as not to repeat the same process. Once a substructure is constructed through the success of a 'goal' call, it should be unnecessary to compute the same job again. Moreover, once it is found that a string of a certain grammar category cannot be obtained from the certain position of the sentence, it is no use to continue the searching process. These are done only by modifying the predicate 'goal' presented in 2.1.

```
goal(G,A,X,Z) :-
        (wf_goal(G,_,X,_);
         fail_goal(G,X),!, fail),!,
         wf_goal(G,A,X,Z)).
goal(G,A,X,Z):-
         dictionary(C,Al,X,Y), link(C,G),
         P=..[C,G,Al,A,Y,Z], call(P),
         assertz(wf_goal(G,A,X,Z)).
goal(G,A,X,Z):-
        (wf_goal(G,_,X,_);
         assertz(fail_goal(G,X))),!, fail.
```

The second definition of 'goal' is the same as the original one except that the successful goal is asserted with name 'wf_goal', which stands for a well formed goal pattern. There are two cases in which the second clause fails, one is when it has never succeeded in finding a well formed goal pattern, and the other is that it has found at least a success goal pattern. In the former case, it is no use to call 'goal' in the same situation. The third definition asserts this fact with the term 'fail_goal'. Note that the only information required here is the current goal G and the parsing position of sentence X. Once 'goal' is executed for goal G and a certain position of the sentence, it need not be tried again. The first definition checks whether 'goal' is called in just the same situation encountered before. The meaning of this clause is as follows: if success goal patterns have been found, it is sufficient to use them, otherwise, if it is known that the trial of calling 'goal' in the given situation eventually fails, it should not go further, and if there is no information about the current calling, the second clause must be executed.

§4 BUP Tools

This section explains two facilities which help users to develop a grammar.

4.1 BUP Translator

It is easier for users to develop grammar rules in DCG syntax instead of the BUP one, because of DCG's perspicuity. The authors consider grammar rules in DCG syntax as a logical expression of grammatical relations, not as a program. The BUP translator gives the grammar an operational semantics (its bottom-up interpretation) by translating DCG descriptions into BUP (Prolog)

programs. The BUP translator reads all the grammar rules in DCG format and performs four tasks. They are the translation of grammar rules into BUP programs, the generation of 'link' clauses, the generation of terminate clauses and the generation of 'public' and 'mode' clauses for grammar rule predicates. The clauses, 'public' and 'mode', are necessary for compiling the BUP programs.

This section explains a BUP translator translation method. While the grammar rules in DCG syntax are categorized into three types, the BUP translator must select its translation method according to the type of grammar rule. Therefore, the rule types and the basic translation methods will now be described.

[1] **Dictionary type rule**

[**Definition**]

A rule containing no nonterminal symbols on the right hand side.

[**Translation principle**]

This rule is translated into an assertion whose predicate is 'dict'. Its first argument is a grammatical category, the second is an argument list in the CFG rule. The third and the fourth arguments construct d-lists. If the CFG rule has some extra conditions, they are put in the body in the same order.

[**Translation example**]

```
DCG : noun(singular, male)➞{pre_cond},[boy],{post_cond}.
BUP : dict(noun,[singular, male],[boy|X],X) :-
                                        pre_cond, post_cond.
```

[2] **Nonterminal type rule**

[**Definition**]

A rule whose leftmost symbol on the right hand side, except extra conditions, is a nonterminal symbol.

[**Translation principle**]

The leftmost nonterminal symbol on the right hand side of the original rule becomes the head of the target clause. The rest of the nonterminal symbols are embedded in 'goal' predicates and the left hand side symbol is put at the tail of the body. Terminal symbols are embedded in the continuation passing mechanism using "=". Predicate 'link' for checking the link relation becomes the top of the body.

[**Translation example**]

```
DCG : npp(X)➞ np(X), {constraint(X)},
                [which],s1(X),[and],s2(X).
BUP : np(G,[X],A,S0,S) :-
                link(npp,G),
                constraint(X),S0=[which|S1],
                goal(s1,[X],S1,S2),S2=[and | S3],
                goal(s2,[X],S3,S4),
                npp(G,[X],A,S4,S).
```

[3] **Terminal type rule**

[**Definition**]

A rule whose leftmost symbol, except extra conditions on the right hand side, is a terminal symbol.

[**Translation principle**]

This case is same as the nonterminal type except that the new nonterminal symbol is generated corresponding to the first terminal symbol and its dictionary item is asserted.

[**Translation example**]

```
DCG : np(np(X)) → {is_sin(X)},[one],noun(X).
BUP : terminal1(G,_,A,S0,S) :-
                        link(np,G),
                        is_sin(X),
                        goal(noun,[X],S0,S1),
                        np(G,[np(X)],A,S1,S).
dict(terminal1,[],[one|X],X).
```

The BUP translator can handle the rule in which symbols are combined with OR(;).[7] For example,

```
np→ np, ([that];[which]), s.
```

is transformed into

```
np(G,[],A,S0,S) :-
                link(np,G),
                (S0=[that|S1]; S0=[which|S1]),
                goal(s,[],S1,S2),
                np(G,[],A,S2,S).
```

4.2 BUP Tracer

In the course of developing and debugging grammar rules, tracing facilities are indispensable. In particular, an interactive debugging environment enables a user to develop grammar rules in a shorter time. This section describes the BUP tracer which provides an interactive debugging environment.

The DEC-10 Prolog system provides very powerful debugging tools such as "trace" and "spy".[1] Although one may debug grammar rules translated into Prolog programs using these tools, it will be a very irritating task for the following reasons:

(1) There is no distinction between grammar rule predicates and extra-predicates.

(2) Since the control of the Prolog interpreter does not correspond with the grammar rule application mechanism, it is difficult to grasp the parsing situation when a backtrack occurs.

In order to solve such problems, the BUP tracer was implemented which displays parsing information and accepts parsing control commands. The BUP

tracer is an augmented Prolog interpreter written in Prolog. The augmentations are to create partial parsing trees and to interact with a user to provide debugging facilities. An interaction occurs when a partial parsing tree is constructed or when some goal is predicted or fails. A user can parse a sentence step by step at a grammar rule level. In each step, the parsing situation (partial parsing trees and applicable rules) is displayed. The main available operations are as follows:

(1) Selection of the grammar rule to apply next.
(2) Skipping the trace until control returns to the current step.
(3) Retry parsing from the current step.
(4) Entering Prolog trace mode.

According to the trace switch control, the facilities that correspond to "spy" in DEC-10 Prolog are implemented.

(1) Entering the trace when the specified grammatical categories are predicted as a goal.
(2) Entering the trace when specified partial parsing trees are created.

Figure 2 shows the example of a CRT-screen in a trace mode. The upper side of the screen shows a partial parsing situation during the current step. The lower side shows currently applicable rules and the input sentence.

```
COMMAND>p

[pg bup wa] [adjvhd iroiro]

---------------------------------------------------------------

===TERM adjvhd(adjvhd(iroiro))===
      1. adjvbp(adjvbp(adjvhd(iroiro),na),nounmod)<--*,[na]
      2. adjvbp(adjvbp(adjvhd(iroiro),ni),verbmod)<--*, [ni]
      3. adjvbp(adjvbp(adjvhd(iroiro),_G),matrix)<--*,([da];[dearu])

---------------------------------------------------------------

Input Sentence
| : bupwa iroirona bunwo kaisekisuru.
```

Fig. 2 A example of trace screen

§5 Discussions

BUP was developed to overcome the following defects of DCG :

(1) DCG cannot deal with left recursive rules.

(2) The grammar and the dictionary are not independently accessible in DCG.

BUP does not have these drawbacks, but it requires that the grammar be cycle-free and have no e-productions. Although a cycle-free grammar is a special case of left recursive grammar, it is not difficult to rewrite a grammar with cycles to a cycle-free grammar.

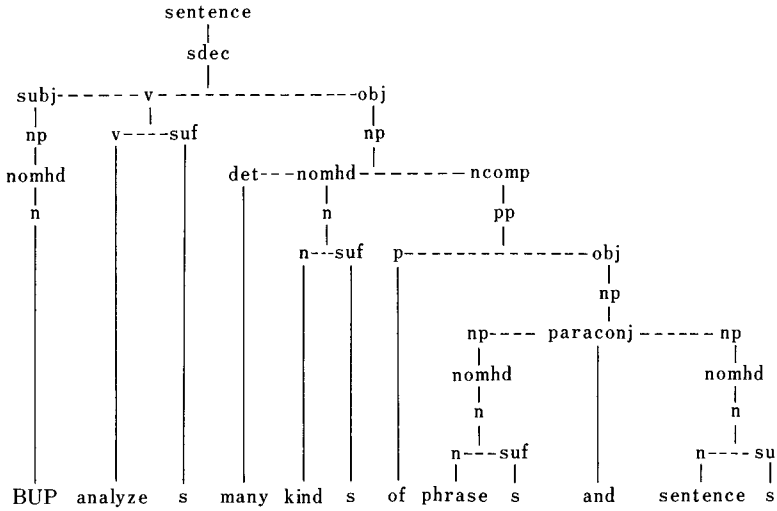
Although current BUP does not accept e-productions, it is easy to augment this system to handle such rules. The DEC-10 Prolog can express "logical or" in a body of a clause. If the given grammar includes an e-production, for example, "C→[]", then replace every occurrence of "goal(c)" in BUP clause bodies with "(goal(c);[])". If a head name of BUP clause is "c", then add another BUP clause that is to be created from the grammar rule the form of which is the same as the original one except that the first category on the right hand side is deleted. Through this modification, it is necessary to check whether it changes the grammar to have cycles or e-productions, again. If so, these processes will be applied repeatedly.

The BUP system requires some additional Prolog clauses (goal, link, and terminate clauses) that DCG does not need. And each nonterminal symbol needs some additional variables in BUP. Thus, the total parsing program is larger in BUP than in DCG. These make BUP less efficient when the parsing process can be done in a deterministic manner. It must be noted, however, that the advantages of BUP may have a more positive effect on execution time. The bottom-up parsing mechanism together with the top-down expectation makes the search space smaller, and looking up the dictionary can be made more efficient in BUP, since it is done by only one predicate, "dict".

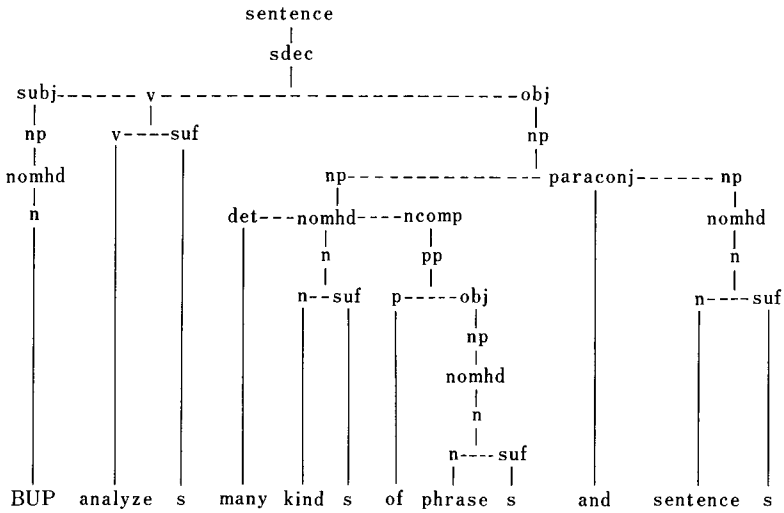
The refinements introduced in Section 3 are peculiar to BUP. Morphological analysis is easily implemented in BUP because looking-up the dictionary is done by the sole predicate, 'dict'. In the same manner, predicates 'wf_goal' and 'fail_goal' guarantee that the task of searching for a string of a given grammar category is not tried repeatedly. This is also easily implemented in BUP because searching for a string is dealt with only by the predicate 'goal'.

Although DCG can be similarly reinforced, that is, it possible for DCG to have a morphological analysis facility and to avoid repeating the same jobs, it is more complicated in DCG than in BUP and efficiency is not improved compared with BUP. This will be discussed elsewhere. The authors are now developing an English grammar with around 100 rules and 200 dictionary entities. The refinements stated in Section 3 makes the BUP system about ten times faster than the original one. Figure 3 shows sample parsing trees analyzed by the refined BUP system.

```
| : BUP alalyzes many kinds of phrases and sentences.
960 msec.
No.1
```




```
485 msec.
No.2
```




```
Total Time=2591 msec.
number of wf_goal was : 21.
number of wf_dict was : 8.
number of fail_goal was : 43.
```

Fig. 3 Sample parsing trees

§6 Conclusions

BUP, a bottom-up parsing system, has been introduced. In this system, grammar rules are embedded in a Prolog like DCG which can handle the dictionary independently of grammar rules. The restriction on the form of grammar rules is much smaller compared with that of DCG. In order to improve efficiency, BUP utilizes the "link" relation to reduce the search space and is improved by avoiding to re-execute the same computation. A BUP translator and BUP tracer have also been developed.

Acknowledgements

Authors wish to express their great gratitude to Mr. Kazuhiro Fuchi, the director of the Research Center of ICOT for his encouragement and comments. Mr. Masaki Kiyono helped us to implement the refinements and gave us various comments.

References

1) Pereira, L., Pereira, F. and Warren, D.: "User's Guide to DEC System-10 Prolog", (Department of Artificial Intelligence, University of Edinburgh, Sept. 1978).
2) Pereira, F. and Warren, D.: "Definite Clause Grammar for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks", Artificial Intelligence, *13* (May, 1980) 231-278.
3) Aho, A. V. and Ullman, J. D.: "The Theory of Parsing, Translation, and Compiling, vol. I Parsing" (Prentice-Hall, 1972).
4) Earley, J.: "An Efficient Context-free Parsing Algorithm" Ph. D. Thesis (Carnegie-Mellon University, 1968).
5) Pratt, V. R.: "A Linguistic Oriented Programming Language", Proc. of 3rd IJCAI (Aug. 1973) 372-381.
6) Pratt, V. R.: "LINGOL — A Progress Report", Proc. of 4th IJCAI (Aug. 1975) 422-428.
7) Miyoshi, H. et. al.: "Bottom-Up DCG Parser User's Manual", ICOT Technical Memo, *No. 6* (1982).