DEFINITE-CLAUSE SET GRAMMARS: A FORMALISM FOR PROBLEM SOLVING*

TAKUSHI TANAKA

▷ First, we present definite-clause set grammars (DCSG), a DCG-like formalism for free-word-order languages. The DCSG formalism is well suited for problem solving. By dealing with DCSG as a generalized parsing problem, we avoid a certain type of looping problem in backward chaining. Next, we extend DCSG by viewing grammar rules as definitions for set conversions. In order to realize inverse conversions, we introduce an inverse operator into DCSG syntax. This operator enables partial bottom-up analyses in the DCSG top-down parsing process. Next, we discuss a looping problem called "left recursion" in top-down parsing. The looping problem is avoided by the bottom-up mechanism of the extended DCSG. The bottom-up mechanism can be viewed as the top-down controlled firing of production rules. Unlike most production systems, production systems written in extended DCSG can backtrack and produce alternative solutions. DCSG is a simple but powerful tool for generalized parsing problems which involve finding structures in a given data set. ◁

1. INTRODUCTION

Definite-clause grammars (DCGs) [11] are a method for expressing context-free grammars in logic programming. A set of grammar rules itself forms a logic program which implements top-down parsing. Inspired by the method of DCG, we have developed a method for structural analysis of electronic circuits which consist of hierarchically organized functional blocks [17]. This method is based on the concept of using difference sets (complementary sets) of circuit elements to define circuit structures, whereas DCG uses difference lists of terminal symbols to define

Address correspondence to Professor Takushi Tanaka, Fukuoka Institute of Technology, 3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-02, Japan.

*This work was supported in part by Grant-in-Aid for Scientific Research of Japanese Ministry of Education.

Received November 1987; accepted October 1988.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/91/$3.50

grammar rules. The difference set method can be viewed as a definite-clause grammar for free-word-order languages.

Here, we first formalize the previously developed method as definite-clause set grammars (DCSGs). In contrast with the usual DCG formalism, DCSG is based on difference sets as opposed to difference lists. DCSG not only treats free-word-order languages, but also is well suited for problem solving. By dealing with DCSG as a generalized parsing problem, we avoid a certain type of looping problem in backward chaining.

Another logic-grammar approach to free-word-order languages is the *gapping-grammar* approach, motivated by linguistics [3, 4]. Its grammar rules allow one to deal with unspecified strings of terminal symbols called gaps. As the gapping-grammar approach explicitly defines permutations of terminal symbols by grammar rules, it is suitable for languages with partially free word order, in which most orderings are fixed. DCSG treats languages with completely free word order, which have up to now been viewed as simple sets, not languages. From a linguistic point of view, DCSG can be viewed as an implementation for the ID rule in GPSG [5], which separates grammar rules into ID (immediate dominance) rules without word order and LP (linear precedence) rules for word order.

Next, we extend DCSG by viewing grammar rules as definitions for set conversions. In order to represent inverse conversions, we introduce an inverse operator into the DCSG syntax. This operator enables partial bottom-up analyses in the top-down parsing mechanism of DCSG.

Next, we discuss a looping problem called "left recursion" in top-down parsing. This problem is avoided by partial bottom-up analysis in extended DCSG. The bottom-up mechanism can be viewed as top-down controlled firing of production rules. Unlike most production systems, production systems written in extended DCSG can backtrack and produce alternative solutions. DCSG is a simple but powerful tool for generalized parsing problems which involve finding structures in a given data set.

Concerning the bottom-up analysis, we also discuss another method originally developed by Matsumoto et al.: BUP [8]. The concept of difference sets is useful for extending not only DCG but also BUP. Using difference sets instead of difference lists, we extend BUP for free-word-order languages.

2. DEFINITE-CLAUSE GRAMMARS FOR FREE-WORD-ORDER LANGUAGES

2.1. *Free-Word-Order Languages*

We can define a free-word-order language $L(G')$ by modifying the definition of formal grammars as follows. We define a context-free free-word-order grammar G' to be a quadruple $\langle V_N, V_T, P, S \rangle$ where V_N is a finite set of nonterminal symbols, V_T is a finite set of terminal symbols, P is a finite set of grammar rules of the form

$$A \longrightarrow B_1, B_2, \ldots, B_n. \qquad (n \geq 1),$$

$$A \in V_N,$$

$$B_i \in V_N \cup V_T \qquad (i = 1, \ldots, n),$$

and S is the starting symbol. The above grammar rule means rewriting a symbol A not with the string of symbols "$B_1, B_2, \ldots, B_n$", but with the set of symbols $\{B_1, B_2, \ldots, B_n\}$. A sentence in the language $L(G')$ is a set of terminal symbols which is derived from S by successive application of the grammar rules. That is, the sentences are multisets which admit multiple occurrences of elements taken from V_T. Each nonterminal symbol used to derive a sentence can be viewed as a name given to a subset of the multiset.

In this study, we actually use a list of elements to represent a multiset. For instance, a free-word-order sentence $\{a, b, c\}$ is represented as one of the following lists: $[a, b, c]$, $[a, c, b]$, $[b, c, a]$, $[b, a, c]$, $[c, a, b]$, $[c, b, a]$.

2.2. *Definite-Clause Set Grammars*

A free-word-order sentence is a multiset of elements taken from V_T. Each nonterminal symbol used to derive the sentence can be viewed as a name given to a subset of the multiset. Therefore, the grammar rules represent relationships between these subsets and elements. Using the predicates "subset" and "member", we can implement definite-clause set grammars for parsing free-word-order languages, analogous to the DCG formalism. We will define a procedure for translating grammar rules to definite clauses. In the present study, both terminal and nonterminal symbols in grammar rules are written as strings of characters beginning with a lowercase letter. Each terminal symbol in a grammar rule is surrounded by "`[`" and "`]`", so that the translation procedure can distinguish the terminal symbol from nonterminal symbols.

The grammar rule which generates nonterminal symbols from a nonterminal symbol

```
s --> np, vp.                                    (1)
```

is translated into a definite-clause form:

```
subset(s,S0,S2) :- subset(np,S0,S1),
                   subset(vp,S1,S2).             (1')
```

The arguments `S0`, `S1`, and `S2` are multisets of V_T, represented as lists of elements. The predicate "`subset`" is used to refer to a subset of an object set which is given as the second argument, while the first argument is a name of its subset. The third argument is a complementary set which is the remainder of the second argument less the first; e.g. "`subset(s,S0,S2)`" states that "`s`" is a subset of `S0` and that `S2` is the remainder. When the clause (1') is used in parsing, an object set is substituted into `S0`, and the nonterminal symbol "`s`" is identified as a subset of `S0`. Procedurally, the clause (1') can be read as follows: In order to show "`s`" to be a subset of `S0`, show that "`np`" is a subset of `S0`, and show that "`vp`" is a subset of `S1` which is the complement of "`np`" in `S0`. `S2` holds the remaining elements after the "`np`" and "`vp`" have been removed.

The grammar rule which generates a terminal symbol from a nonterminal symbol

```
noun --> [book].                                 (2)
```

is translated into a definite-clause form:

```
subset(noun,S0,S1) :- member(book,S0,S1).        (2')
```

The clause (2') can be read as follows: In order to show "noun" to be a subset of S0, show "book" to be a member of S0, and set the remainder to S1. The predicate "member" is defined as follows:

```
member(M,[M|X],X).
member(M,[A|X],[A|Y]) :- member(M,X,Y).          (3)
```

The predicate "member" has three arguments. The first is an element of a set. The second is the whole set. The third is the complementary set of the first. That is, both terminal and nonterminal symbols are represented as differences of the last two arguments of these predicates.

The general form of the translation procedure from a grammar rule

$A \longrightarrow B_1, B_2, \ldots, B_n.$

to a definite clause is

$$\begin{aligned} \texttt{subset}(A, S_0, S_n) \texttt{ :- } & \texttt{subset}(B_1, S_0, S_1), \\ & \texttt{subset}(B_2, S_1, S_2), \\ & \vdots \\ & \texttt{subset}(B_n, S_{n-1}, S_n). \end{aligned}$$

Here, all symbols in the grammar rule are assumed to be nonterminal symbols. If "[B_i]" ($1 \leq i \leq n$) is found in the right-hand side of grammar rules, where "B_i" is assumed to be a terminal symbol, then "member(B_i, S_{i-1}, S_i)" is used instead of "subset(B_i,S_{i-1},S_i)" in the translation.

Several context-dependent extensions to the basic context-free formalism are necessary for actual use. For example, "test C" and "not C" in grammar rules are context-dependent conditions which respectively demand the existence and the absence of subset C in the current parsing context. The forms "test C" and "not C" in

$A \longrightarrow B_1, \ldots, B_i,$ test $C, B_{i+1}, \ldots, B_n.$

$A \longrightarrow B_1, \ldots, B_i,$ not $C, B_{i+1}, \ldots, B_n.$

are respectively translated into

subset(C,S_i,_)

and

not subset(C,S_i,_).

The "_"s are anonymous variables which may be ignored. Conditions of the form "test [C]" and "not [C]" will be translated into

member(C,S_i,_)

and

not member(C,S_i,_),

which respectively demand the existence and the absence of element C in the

current context S_i. Other extensions to the basic DCSG formalism are explained in later sections.

2.3. *Generation*

We will consider the problem of generating free-word-order sentences as list expressions. As we have already defined DCSG for parsing, first we will examine the inverse process of DCSG parsing, using the following example.

The DCSG-translation procedure changes the grammar rule

```
s --> [a], [b], [c].                                    (4)
```

into a definite-clause form:

```
subset(s,S0,S3) :- member(a,S0,S1),
                   member(b,S1,S2),
                   member(c,S2,S3).                     (4')
```

The definite clause (4') can successfully parse all permutations of `a`, `b`, and `c`, namely, `[a,b,c]`, `[a,c,b]`, `[b,c,a]`, `[b,a,c]`, `[c,a,b]`, `[c,b,a]`, which are list expressions of the set {`a`, `b`, `c`}. But the following goal clause, which exchanges input and output in the predicate "`subset`", cannot generate all of these permutations as would be expected in a DCG-based system:

```
?- subset(s,S0,[ ]).
```

The goal clause loops after generating the two answers as `S0 = [a,b,c]`; `[a,c,b]`. The first answer is a result of the following unifications: `S0 = [a|S1]`, `S1 = [b|S2]`, `S2 = [c|S3]`, and `S3 = [ ]`. As the third subgoal has a unique result `S2 = [c]`, the second answer is a result of backtracking of the second subgoal unified as `S1 =` $[X_1,b|X_2]$ and `S2 =` $[X_1|X_2]$, namely, X_1 = `c`, X_2 = `[ ]`. Looking for the third answer, the second subgoal backtracks as `S1 =` $[X_1,X_2,b|X_3]$, `S2 =` $[X_1,X_2|X_3]$, but these unifications conflict with `S2 = [c]` of the third subgoal; then the second subgoal backtracks. Goals, such as "`member(b,S1,S2)`", in which the second and the third arguments are both variables, have an infinite number of answers: `member(b,`$[X_1,\ldots,X_{n-1},b|X_n]$, $[X_1,\ldots,X_{n-1}|X_n]$`)`. Thus, the inverse process does not work successfully for generating all permutations.

In order to generate the permutations, we must avoid the infinite answers which are caused by the remaining variables. This can be done by defining a separate DCSG-translation procedure for generation. There are two ways to do this. One method is to exchange the second and the third arguments in each goal:

```
subset(s,S3,S0) :- member(a,S1,S0),
                   member(b,S2,S1),
                   member(c,S3,S2).
```

Another method is to reverse the order of subgoals in the right-hand side:

```
subset(s,S0,S3) :- member(c,S2,S3),
                   member(b,S1,S2),
                   member(a,S0,S1).
```

According to the first method, the following goal clause generates all permutations of a, b, and c:

```
?- subset(s,S3,[ ]).
```

The first subgoal succeeds with a unique answer `S1 = [a]`. The second subgoal generates all permutations of a and b, namely `[a,b]` and `[b,a]`, as `S2`. The third subgoal generates all permutations of a, b, and c as `S3` by adding c into all possible positions of `[a,b]` and `[b,a]`.

3. PROBLEM SOLVING AS GENERALIZED PARSING

3.1. Backward Chaining and Top-Down Parsing

In order to show the advantages of using DCSG in problem solving, we first compare backward chaining and top-down parsing of free-word-order language.

Usually, a logic program consists of two kinds of definite clauses, called facts $\{F_1, F_2, \ldots, F_n\}$ and rules $\{R_1, R_2, \ldots, R_m\}$. Both kinds are viewed as axioms. Computations based on refutation can be viewed as a process of deriving a theorem by backward chaining from the axioms. The top-down parsing of a free-word-order sentence somewhat resembles the process of backward chaining. The object sentence is given as a set of terminal symbols $\{W_1, W_2, \ldots, W_n\}$. The starting symbol "$S$" is decomposed into terminal symbols using grammar rules $\{G_1, G_2, \ldots, G_m\}$ until they coincide with the given sentence. That is, the set of facts corresponds to the sentence, and the set of backward chaining rules corresponds to the set of grammar rules. Deriving theorems in backward chaining corresponds to identifying nonterminal symbols in the top-down method.

There is an important difference between backward chaining and top-down parsing. Backward chaining allows multiple use of the same fact to derive a theorem, while in a context-free language, each terminal symbol in a sentence contributes only once to the reduction of nonterminal symbols. This characteristic is very useful for avoiding a common looping problem in backward chaining, the problem which is caused by multiple use of the same fact. In the following sections, we will see an example of this looping problem, and how to solve it by viewing problem solving as a generalized parsing problem in DCSG.

3.2. The Looping Problem

When problems are formalized and expressed in logic programming, we often encounter a certain kind of looping problem. We will clarify a cause of this problem using the example of the problem of voltage derivation.

Assume that voltages on a circuit are as given in Figure 1. We might consider representations of the voltage data by the following assertions:

```
voltage($1,$2,20).
voltage($3,$2,15).                    (5)
voltage($3,$4,8).
```

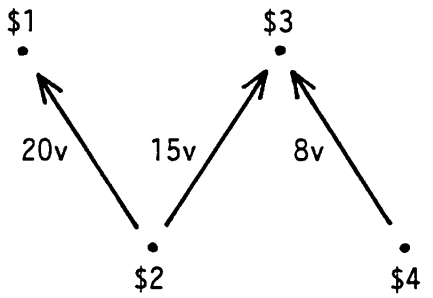


FIGURE 1. Voltages on a circuit.

"`voltage($1,$2,20)`" states that the voltage between node `$1` and node `$2` is 20 volts. In order to derive the voltage data independently of the node order, we could consider defining "`volt`" by the following clauses:

```
volt(A,B,V) :- voltage(A,B,V).
volt(A,B,-V) :- voltage(B,A,V).
```
(6)

Furthermore, we will define the predicate "v" that derives voltages between two arbitrary nodes `A` and `C` as

```
v(A,C,V) :- volt(A,C,V).
```
(7)

```
v(A,C,V+W) :- volt(A,B,V), v(B,C,W).
```
(8)

But these definitions will not work as intended. In order to derive the voltage between `$1` and `$4`, we attempt to execute the following goal clause:

```
?- v($1,$4,X).
```

As the voltage between `$1` and `$4` is not given, the goal is decomposed into subgoals by (8). The first subgoal succeeds by "`volt($1,$2,20)`", binding node `B` with `$2`. The second subgoal "`v($2,$4,W)`" is also decomposed into subgoals by (8). The first subgoal succeeds as "`volt($2,$1,-20)`" using the same voltage data, and the second subgoal becomes the same as the initial goal. Thus, the system loops.

One method to avoid this problem is to erase voltage data as they are used, so that the same datum is not used twice. This can be done by replacing (6) with

```
volt(A,B,V) :-  voltage(A,B,V),
                retract(voltage(A,B,V)).
volt(A,B,-V) :- voltage(B,A,V),
                retract(voltage(B,A,V)).
```
(6′)

But the erased data cannot be recovered in backtracking.

Another common method keeps track of the data used, so that the same datum is not used twice. Replacing (7) and (8) with (7′) and (8′), we acquire the voltages between nodes `$1` and `$4` by the goal clause "`?- v($1,$4,X,[ ]).`". This method has the disadvantage of requiring the overhead of explicitly keeping track

of the data used:

```
v(A,C,V,_) :- volt(A,C,V).                                    (7')
v(A,C,V+W,T) :- volt(A,B,V),
                not member(B,T,_),
                v(B,C,W,[A|T]).                               (8')
```

In the next section, we show how to avoid this problem by viewing problem solving as a generalized parsing problem.

3.3. Solution

To solve the above looping problem, we introduce a change of representation which involves viewing the voltage derivation problem not as a backward search problem, but as a parsing problem. This involves a change of representation of the node-voltage data from assertions to a set of function-argument terms. Each expression "`voltage(A,B,V)`" forms a compound term. The voltages of Figure 1 are represented by a set of those terms using a list:

```
vData([voltage($1,$2,20),voltage($3,$2,15),
       voltage($3,$4,8)]).                                    (9)
```

Accordingly, we represent the voltage derivation not as clauses for backward chaining but as grammar rules for parsing. The following grammar rules correspond to the clause (6):

```
volt(A,B,V) --> [voltage(A,B,V)].
volt(A,B,-V) --> [voltage(B,A,V)].                            (10)
```

"`voltage(A,B,V)`" surrounded by "`[`" and "`]`" is a terminal symbol, while "`volt(A,B,V)`" is a nonterminal symbol. Here, we have introduced universally quantified variables (A, B, and V) into the grammar rules. These variables are instantiated when they are applied to object sentences. According to the DCSG translation procedure, the grammar rules are translated into the following clauses:

```
subset(volt(A,B,V),S0,S1):-
          member(voltage(A,B,V),S0,S1).
   subset(volt(A,B,-V),S0,S1):-
          member(voltage(B,A,V),S0,S1).                       (10')
```

In order to derive the voltage between two arbitrary nodes, we define the following grammar rules corresponding to (7) and (8):

```
v(A,C,V) --> volt(A,C,V).                                     (11)
v(A,C,V+W) --> volt(A,B,V), v(B,C,W).                         (12)
```

The grammar rules are translated into

```
subset(v(A,C,V),S0,S1) :- subset(volt(A,C,V),S0,S1).          (11')
subset(v(A,C,V+W),S0,S2) :- subset(volt(A,B,V),S0,S1),
                            subset(v(B,C,W),S1,S2).           (12')
```

Deriving the voltage X between nodes $1 and $4 is accomplished by identifying the nonterminal symbol "v($1,$4,X)" in the free-word-order sentence (9) as follows:

```
?- vData(VD),
   subset(v($1,$4,X),VD,_).
X = 20 + ( -15 + 8)
```

Terminal symbols associated with the nonterminal symbol "v($1,$4,X)" are removed sequentially from the object sentence. Therefore, the looping problem due to using the same data repeatedly does not occur. This method is like that described in (6') in the previous section. But the removed data can be recovered in case of backtracking.

We have overcome a common looping problem in backward chainings by simply rewriting backward chaining rules into grammar rules, and by changing a set of facts into a free-word-order sentence. The looping problem which is caused by multiple use of the same facts is avoided by dealing with DCSG as a generalized parsing problem.

4. EXTENDED DCSG

4.1. Set Conversion

A terminal symbol "[voltage(A,B,V)]" in the grammar rules was translated into "member(voltage(A,B,V),S0,S1)" by the DCSG translation procedure. When an object set is substituted into S0, the element "voltage(A,B,V)" is removed from the set, and the remainder is bound to the variable S1. Therefore, we can view a terminal symbol in grammar rules as a function for set conversion which removes an element from an object set (Figure 2). On the other hand, a nonterminal symbol works as a set conversion which removes a subset of the object set, since nonterminal symbols are finally defined by a combination of terminal symbols. Therefore, we can view grammar rules as definitions for set conversion.

Here, we consider the inverse of the set conversion. The inverse conversion for a terminal symbol is adding the symbol itself. The inverse conversion for a nonterminal symbol is adding a set of terminal symbols derived from the nonterminal symbol. Though we have already developed methods of generating free-word-order sentences in Section 2.3 as the inverse process of parsing, the methods are not suitable for this purpose of inverse set conversion. For instance, when we use

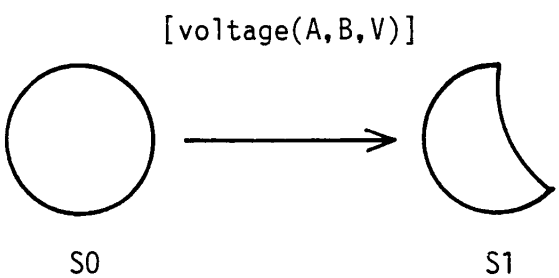


FIGURE 2. Set conversion removing an element.

the method for the terminal symbol "m" as in

```
?- member(m,OUT,[a,b,c]).,
```

four cases occur: OUT = [m,a,b,c], [a,m,b,c], [a,b,m,c], [a,b,c,m]. This is because lists are ordered, but are used to represent sets, which are not ordered. These permutations cause spurious ambiguities for inverse set conversion.

4.2. Extended DCSG with Inverse Operator

We will extend DCSG by viewing grammar rules as definitions for set conversions. In order to represent inverse conversions, we introduce an inverse operator into DCSG syntax. A terminal symbol prefixed by the inverse operator as "add $[B_i]$" $(1 \leq i \leq n)$ in

$$A \rightarrow B_1, \ldots, B_{i-1}, \text{ add } [B_i], B_{i+1}, \ldots, B_n.$$

is translated into a formula

$$S_i = [B_i \mid S_{i-1}].$$

That is, "add" directs addition of the following symbol into the object set S_{i-1} to make a new set S_i. Thus, a set conversion for adding an element is defined uniquely.

The inverse conversion for a nonterminal symbol B_i is represented as "add B_i", which means adding a set of terminal symbols derived from B_i. This generation is attained by the inverse process of ordinary DCG, rather than the methods in Section 2.3, because DCG generates a unique list of terminal symbols, while DCSG generates their permutations.

Nonterminal symbols defined by a grammar rule with the inverse operator are no longer viewed as names given to subsets of an object set. As the predicate "subset" is inadequate, we use the predicate "convert" for nonterminal symbols instead of "subset" in extended DCSG. The inverse operator enables partial bottom-up analyses in the DCSG top-down parsing process.

5. BOTTOM-UP ANALYSIS

5.1. Difficulties in Top-Down Parsing

In order to show the advantages of the extended DCSG, we first discuss a looping problem called *left recursion*. The top-down parsing mechanism of DCSG loops on left-recursive rules, i.e. rules which have the same nonterminal symbol as the left-hand side in their leftmost position on the right-hand side. In the next section we solve this looping problem by the bottom-up mechanism in extended DCSG. The following is an example of this looping problem.

A directed graph in Figure 3 is represented as the following data set:

```
gData([arc(a,$1,$2),arc(b,$2,$3),
       arc(c,$3,$4),arc(d,$2,$4)]).        (13)
```

"arc(a,$1,$2)" represents an arc with label "a" connecting nodes $1 and $2. The following grammar rules define a nonterminal symbol sp(X,A,B), which

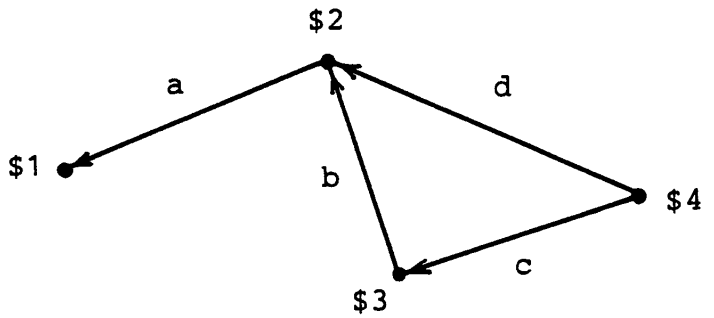


FIGURE 3. A directed graph.

represents a class of graphs called series-parallel connection of arcs between nodes A and B:

```
sp(X,A,B) --> [arc(X,A,B)].                                    (14)
sp(pr(X,Y),A,B) --> sp(X,A,B), sp(Y,A,B).                      (15)
sp(sr(X,Y),A,C) --> sp(X,A,B), sp(Y,B,C),
                    not sp(_,B,_), not sp(_,_,B).              (16)
```

(14) defines a single arc `arc(X,A,B)` between nodes A and B as a nonterminal symbol "`sp(X,A,B)`". (15) defines a parallel connection of `sp(X,A,B)` and `sp(Y,A,B)` as `sp(pr(X,Y),A,B)`. "`pr(X,Y)`" is a label given to the parallel connection. (16) defines a series connection of `sp(X,A,B)` and `sp(Y,B,C)` as `sp(sr(X,Y),A,B)`. "`not sp(_,B,_)`" and "`not sp(_,_,B)`" express the condition that no arc other than X and Y should connect to the central node B in series connection. In order to identify a series-parallel connection between nodes `$1` and `$4` in Figure 3, we attempt the following goal:

```
?- gData(GD),
   subset(sp(X,$1,$4),GD,REST).                                (17)
```

But the second goal loops. The top-down mechanism of DCSG decomposes the starting symbol into terminal symbols iteratively until it generates a set of terminal symbols which coincides with the object data set. As the series-parallel connection is defined by left recursion in (15) and (16), the system infinitely decomposes the starting symbol with the same symbol when the generated elements do not coincide. The starting symbol `sp(X,$1,$4)` is first decomposed into a terminal symbol `arc(X,$1,$4)` by (14). But the symbol does not exist in the object set. Then, the starting symbol is decomposed into the same symbols by (15). Thus the system loops.

In top-down parsing, we must avoid grammar rules with left recursion. In ordinary context-free grammars, this can be done by introducing additional nonterminal symbols to change rules into Greibach normal form [6]. But we cannot use this procedure straightforwardly, because we have introduced variables into grammar rules. Since rules with variables assume an infinite set of terminal symbols, the rules exceed the definition of context-free grammars.

5.2. *Bottom-Up Analysis by Extended DCSG*

We can avoid the looping problem of left recursion by embedding a bottom-up method in the top-down mechanism of DCSG. Bottom-up methods replace lower-level symbols by higher-level symbols until a sentence is replaced with a starting symbol. We can realize this process as set conversions using the extended DCSG. The key idea is actually rewriting terminal symbols into nonterminal symbols in the object set.

First, we change grammar rules into rules for bottom-up analysis, which we call one-step reductions. The nonterminal symbol in the left-hand side is changed into a pseudo terminal symbol prefixed by the inverse operator, and moved to the rightmost position of the right-hand side. According to this procedure, the grammar rule (14) is changed into (14′) below. The left-hand side is given a new nonterminal symbol "`rule1`" as

```
rule1 --> [arc(X,A,B)], add [sp(X,A,B)].        (14′)
```

The rule is translated into a definite-clause form:

```
convert(rule1,S0,S2) :- member(arc(X,A,B),S0,S1),
                        S2 = [sp(X,A,B)|S1].
```

The nonterminal symbol "`rule1`" is a set conversion which first removes an element `arc(X,A,B)`, then adds a new element `sp(X,A,B)`. The element `sp(X,A,B)` is a pseudo terminal symbol. It is grammatically a nonterminal symbol, but it is also a real member of the object set; therefore it is procedurally treated as a terminal symbol. We define the following rules in place of (15) and (16):

```
rule2 --> [sp(X,A,B)], [sp(Y,A,B)],
          add [sp(pr(X,Y),A,B)].                 (15′)
rule3 --> [sp(X,A,B)], [sp(Y,B,C)],
          not [sp(_,B,_)], not [sp(_,_,B)],
          add [sp(sr(X,Y),A,C)].                 (16′)
```

The nonterminal symbol "`rule2`" is a set conversion which first removes two elements connected in parallel, then adds a new element. The nonterminal symbol "`rule3`" is a set conversion for series connections. Each rule represents a one-step reduction from lower-level symbols to a higher-level symbol.

In order to realize the bottom-up analysis, we must iterate these reductions until the object set is reduced to a starting symbol. There are several methods to iterate these reductions. The following method separates the reduction process into two stages, so that the process avoids unnecessary backtracking. (18) and (19) define "`stage1`" as an iteration of `rule1, stage1` which rewrites all of the input symbols into pseudo terminal symbols:

```
stage1 --> rule1, stage1.                        (18)
stage1 --> not rule1.                            (19)
```

(18) iterates `rule1` until no input symbol remains in the object set. When (18)

fails, (19) is applied, which is translated into

```
convert(stage1,S0,S0) :- not convert(rule1,S0,_).      (19')
```

This clause returns `S0` unchanged.

The following rules (20), (21), and (22) define "`stage2`" as a multiple application of `rule2` and `rule3`. The connective "`;`" is used for abbreviation of two rules which have the same left-hand side. (23) defines the total reduction process. The cut symbol "`!`" is available in DCSG syntax:

```
rule23 --> rule2; rule3.                               (20)

stage2 --> rule23.                                     (21)

stage2 --> rule23, stage2.                             (22)

reduction --> stage1, !, stage2.                       (23)
```

Now, we can identify the given data set as a free-word-order sentence derived from the starting symbol "`sp(X,$1,$4)`":

```
?- gData(GD),
   convert(reduction,GD,[sp(X,$1,$4)|REST]).
X = sr(a,pr(sr(b,c),d))
REST = [ ]
```

The value of `X` keeps track of successful goals. It can be viewed as a parse tree for the graph in Figure 3 (Figure 4).

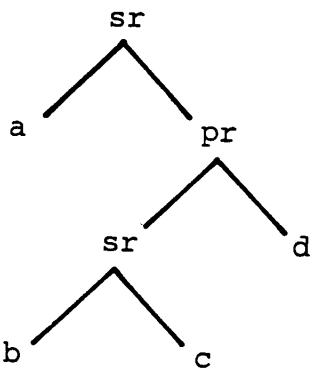


FIGURE 4. Parse tree.

5.3. Another Approach: Extended BUP

Another bottom-up method, BUP [8], was developed for ordinary context-free languages. BUP realizes a data-driven process which first identifies a category of input symbols by dictionary, then derives rules to be used by assuming the category as the leftmost symbol in the right-hand side of grammar rules. We will examine this method by extending BUP for free-word-order languages. The extended BUP is based on the concept of difference sets instead of difference lists. The extended BUP changes the previous rule (14), which generates a terminal symbol from a

nonterminal symbol, into

```
dictionary(sp(X,A,B),S0,S1) :-
                member(arc(X,A,B),S0,S1).          (14″)
```

Other grammar rules which generate only nonterminal symbols

$A \longrightarrow B_1, B_2, \ldots, B_n$.

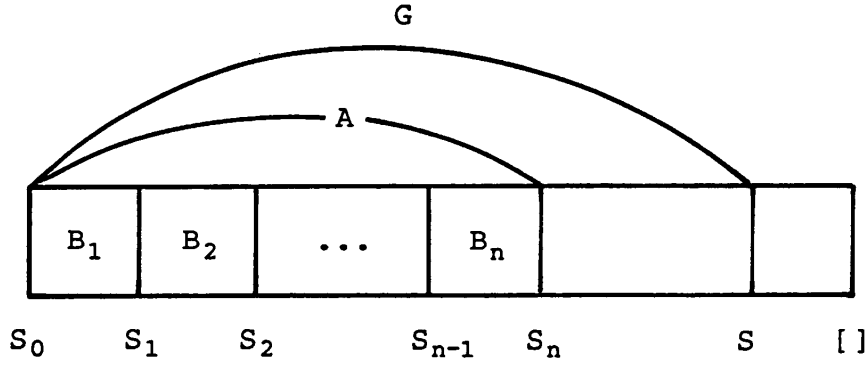
are translated into the following clause form:

```
goal(B1,G,S1,S) :- subset(B2,S1,S2),
                   ⋮
                   subset(Bn,Sn-1,Sn),
                   goal(A,G,Sn,S).
```

The clause can be read: In order to show that the nonterminal symbol B_1 and the difference set "$S_1 - S$" form a nonterminal symbol G, show that $B_2, \ldots, B_n$ are exclusive subsets of S_1, and show that the nonterminal A and the difference sets "$S_n - S$" form the nonterminal symbol G (Figure 5). According to this translation procedure, the grammar rules (15) and (16) are translated into

```
goal(sp(X,A,B),G,S1,S) :-
                subset(sp(Y,A,B),S1,S2),
                goal(sp(pr(X,Y),A,B),G,S2,S).      (15″)
goal(sp(X,A,B),G,S1,S) :-
                subset(sp(Y,B,C),S1,S2),
                not subset(sp(_,B,_),S2,_),
                not subset(sp(_,_,B),S2,_),
                goal(sp(sr(X,Y),A,C),G,S2,S).      (16″)
```

FIGURE 5. Relationships between difference sets.



BUP needs two more clauses as follows:

```
subset(G,S0,S) :- dictionary(B1,S0,S1),
                  goal(B1,G,S1,S).          (24)

goal(A,A,S,S).                              (25)
```

(24) can be read as follows: In order to show that G is a subset of S0 and that S is the remainder, first identify a nonterminal symbol B1 in S0 by the dictionary and set the remainder into S1; then show that B1 and the difference set "S1 - S" forms the nonterminal symbol G (Figure 5). The second subgoal "goal(B1,G,S1,S)" is decomposed by using one of the clauses translated from grammar rules. (25) is a special clause for termination. Using these clauses, the graph in Figure 3 is successfully parsed as follows:

```
?- gData(GD),
   subset(sp(X,$1,$4),GD,REST).
X = sr(a,pr(sr(b,c),d))
REST = [ ]
```

Thus, the concept of the difference set is also useful for extending BUP.

The extended BUP based on difference sets does not work as efficiently as ordinary BUP. The initial goal is decomposed into "dictionary" and "goal" by (24). The "dictionary" identifies a category of the first input symbol of the object set. But the category may not appear in the leftmost position in the right-hand side of any grammar rules. In that case, the "dictionary" must backtrack to look for another symbol with a desired category. In ordinary BUP, this case does not occur unless the input sentence is ungrammatical. The first symbol in each phrase is generated by the leftmost symbol in each grammar rule corresponding to the phrase, so the input symbol directly derives the rule to be used.

5.4. Set Conversion and Production System

We have developed two bottom-up methods (extended DCSG and extended BUP) for free-word-order language. In the extended DCSG method, the user must define the control of the grammar rules, whereas the BUP method directly derives grammar rules to be used from the input string. But the user-definable control is in turn a merit for problem-solving applications. We can easily define production systems in extended DCSG.

The set conversions (14'), (15'), and (16') which define one-step reductions for bottom-up analysis can be viewed as a production rule of the form

ruleName $\longrightarrow$ *condition*,
action.

The object set corresponds to the working memory of a production system [1, 10]. However, this differs from ordinary forward chaining in that elements which have been used to test the condition are removed from the object set if the rule succeeds. The action consists of adding a new element to the object set. The set conversions from (18) to (23) can be viewed as defining a hierarchy of control rules

which determine the order of the productions:

ruleName ⟶ *ruleName*,
ruleName.

These different kinds of rules are consolidated into the same set-conversion syntax, and translated into definite clauses. Unlike most production systems, production systems written in extended DCSG can backtrack and produce alternative solutions. Normal production systems enumerate all choices and pick one by a "conflict resolution" algorithm.

We can also realize ordinary forward chainings as set conversions. The following set conversion implements a forward chaining rule for "every man is mortal":

```
ruleMM --> test [man(X)],
           not [mortal(X)],
           add [mortal(X)].
```

When the object set initially has an element "`man(socrates)`", the set conversion make a new set with elements both "`man(socrates)`" and "`mortal(socrates)`". As the condition part is prefixed by "`test`" in Section 2.2, the condition part does not remove elements which have been used to test the condition. The second condition "`not [mortal(X)]`" prevents unnecessary deductions in forward chainings. The rule lacking this condition generates "`mortal(socrates)`" repeatedly from "`man(socrates)`" every time the rule is called.

6. CONCLUSIONS

DCSG is a simple but powerful tool for generalized parsing problems which involve finding structures in a given data set. The data are represented by compound terms. The set of those compound terms is viewed as a sentence of a free-word-order language. Rules for finding structures are represented as grammar rules. By dealing with DCSG as a generalized parsing problem, we have shown how to avoid a common looping problem in backward chaining caused by multiple use of the same data.

We have extended DCSG by viewing grammar rules as definitions for set conversions. In order to realize the inverse conversion, we introduced an inverse operator into DCSG syntax. The inverse operator changes the characteristics of grammar rules. Nonterminal symbols no longer correspond to a subset of the object set. The nonterminal symbols became names representing set conversions.

Using this inverse operator, we embedded a bottom-up mechanism in the top-down parsing process of DCSG. The bottom-up method avoided the looping problem in top-down parsing. The bottom-up analysis of series-parallel directed graphs can be considered as a production system whose production rules are controlled in a top-down manner. The production rules and their controls were defined in extended DCSG and translated into a logic program. That is, the idea of set conversion enables us to consolidate these different kinds of rules into a uniform syntax.

Concerning bottom-up analysis, we have also extended BUP for free-word-order languages. The concept of difference sets is useful for extending not only DCG but

also BUP. As we have developed two bottom-up methods for parsing free-word-order language, we must further examine these methods on various problems to compare their advantages and disadvantages.

The extended DCSG may apply not only for parsing problems but also for event-state-changing problems such as robot planning. The initial state is represented as a set of compound terms. Each state-changing event is defined as a nonterminal symbol which represents a set conversion. A string of nonterminal symbols represent a sequence of events. We can simulate a sequence of changing states by treating the events as set conversions.

This study was originally developed on a deductive system called Duck. I would like to thank Professor Drew McDermott for Duck. I would like to thank Professor Jacob Mey and Mr. David Littleboy for their helpful advice, and the AIUEO AI circle for useful discussions.

REFERENCES

1. Charniak, E. and McDermott, D., *Introduction to Artificial Intelligence*, Addison-Wesley, Reading, Mass., 1985.
2. Colmerauer, A., Metamorphosis Grammars, in: L. Bolc (ed.), *Natural Language Communication with Computers*, Springer-Verlag, Berlin, 1978.
3. Dahl, V. and Abramson, H., On Gapping Grammars, in: *Proceedings of the 2nd International Conference of Logic Programming*, Uppsala, Sweden, 1984, pp. 77–88.
4. Dahl, V., More on Gapping Grammars, in: *Proceedings of FGCS*, Tokyo, 1984, pp. 669–677.
5. Gazder, G., Klein, E., Pullum, G., and Sag, I., *Generalized Phrase Structure Grammar*, Basil Blackwell, Oxford, 1985.
6. Greibach, S. A., A New Normal Form Theorem for Context-Free Phrase Structure Grammars, *J. Assoc. Comput. Mach.* 12:42–52 (1965).
7. Kowalski, R., *Logic for Problem Solving*, North Holland, New York, 1979.
8. Matsumoto, Y. et al., BUP: A Bottom-Up Parser Embedded in Prolog, *New Generation Comput.* 1:145–158 (1983).
9. McDermott, D., *Duck Reference Manual*, Dept. of Computer Science, Yale Univ., 1983.
10. Nilsson, N. J., *Principles of Artificial Intelligence*, Tioga, Palo Alto, Calif., 1980.
11. Pereira, F. C. N. and Warren, D. H. D., Definite Clause Grammars for Language Analysis, *Artificial Intelligence* 13:231–278 (1980).
12. Pereila, F. C. N., Extraposition Grammar, *AJCL* 7:243–256 (1981).
13. Popowich, F., Unrestricted Gapping Grammars, in: *Proceedings of IJCAI-85*, Los Angeles, Calif., 1985, pp. 765–768.
14. Saint-Dizier, P., Contextual Discontinuous Grammars, in: *Proceedings of the 2nd International Workshop on Natural Language Understanding and Logic Programming*, 1987, pp. 17–19.
15. Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, Cambridge, Mass., 1986.
16. Tanaka, T., Representation and Analysis of Electrical Circuits in a Deductive System, in: *Proceedings of IJCAI-83*, Karlsruhe, W. Germany, 1983, pp. 263–267.
17. Tanaka, T., Parsing Circuit Topology in a Deductive System, in: *Proceedings of IJCAI-85*, Los Angeles, Calif., 1985, pp. 407–410.
18. Tanaka, T., Structural Analysis of Electronic Circuits in a Deductive System, in: L. Bolc and M. J. Coombs (eds.), *Expert System Applications*, Springer-Verlag, Berlin, 1988, pp. 257–308.