ソフトウェア工学 46－6
（１９８６．２．６）

LOGICAL PROGRAMMING FOR THE TELEGRAM ANALYSIS PROBLEM

鳥居宏次，嵩　忠雄，杉山裕二，°森沢好臣
（大阪大学）　　（日本ユニバック）

Abstract

The telegram analysis problem posed by P.Henderson and R.A.Snowdon has been repeatedly taken into account. This paper adds yet another contribution to this problem. We propose rigorous specification methods, and describe how programs can be derived from BNF to Definite Clause Grammar in Prolog in two different methods. One of them is especially useful for a large scale problem, which has been applied to file manipulation, and the other is for a simple problem.

1. Introduction

The rigorous methods to obtain a correct program are still central concerns in Software Engineering. Focusing on the telegram analysis problem [1,2,3,4,5,6], we have investigated several specification and design methods.

In this paper we propose rigorous specification methods and show how programs can be derived from a specification through a clear process. These rigorous approaches are systematic programming methods for obtaining a correct program. The usage of the telegram analysis problem as an example in this paper has no special meaning other than that this example is widely known for more than 10 years.

In Section 2, we describe the definition of the Definite Clause Grammar(abbreviated as DCG) in Prolog.

In Section 3, we restate to clarify the telegram analysis problem.

In Section 4, we propose two rigorous methods for transforming a problem specification in Backus-Naur Form (abbreviated BNF) into a Prolog program through DCG which can be translated into Prolog [7]. The first method starts with writing the input and output data structures in BNF. As is well known, an input into DCG is automatically transformed into a syntax analysis tree. Therefore there is no need to analyse the input explicitly. Further, explicit syntactical generation of the output data structure is not needed. The method is one of the easiest ways for rapid prototyping, since it is suffice to consider the non-terminal symbols in CFG for the syntactical parts. As to the semantic parts the attributes which easily lead to the construction of DCG, relating to each non-terminal, can be derived from the output BNF.[9] We have applied this method to file manipulation. The second method is basically same as the first one except the introduction of the intermediate file, the output data structure and the usage of the attribute grammar for the input's BNF.

2. DCG in Prolog

When we consider the specification in BNF as in [6], we can propose a natural way to derive a Prolog program using DCG. A DCG is a natural extension of CFG. It provides not only a description of a language, but also an effective means for analyzing strings of that language, since DCG is an executable program of the programming language Prolog. Using a standard Prolog compiler, DCG can be compiled into Prolog, making it feasible to implement practical language analysers directly as DCGs[7,8].

DCG is defined in [7] as follows.

A grammar rule in DCG takes the general form:

```
LHS --> RHS.
```

meaning "a possible form for LHS is RHS". Both RHS and LHS are sequences of one or more items linked by the conjunction operator ','.

The DCGs extend context-free grammars in the following ways:

(1) The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals.

(2) The right-hand side of a grammar rule consists of a sequence of non-terminals or terminals.

(3) A non-terminal symbol may be any Prolog term (other than a variable or integer).

(4) A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. If the terminal symbols are ASCII character codes, such lists can be written as strings. An empty sequence is written as the empty list, [] or "".

(5) Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in {} brackets.

(6) Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator ';' as in Prolog.

3. The Telegram Analysis Problem

The original telegram analysis problem posed by P.Henderson and R.A.Snowdon [Hend72] tells:

> A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer area where it is to be processed. The words in the telegrams are separated by sequences of blanks and each telegram is delimited by the word "ZZZZ". The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words "ZZZZ" and "STOP" are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is {{to be a neat listing of the telegrams, each accompanied by}} the word count and a message indicating the occurrences of an overlength word.

When we try to analyse this problem, there are many difficulties from the outset. The above intuitive description has several ambiguities. Since we prefer to analyse the problem under the same environment, we have fixed the problem into a standard in the following manner:

(S1) The phrase in {{...}} is deleted.
(S2) The "STOP" word is dealt with as a special word, i.e., it is not counted as a chargeable word.
(S3) An overlength word is counted as one word, but it is not necessary to count the number of their occurrence.
(S4) The telegram number is the serial number of the telegram in the telegram sequence.

Therefore, for example, the output of this problem has a form described as follows:

```
Telegrams Analysis
Telegram 1. Charge count is 15, overlength occurred.
Telegram 2. Charge count is 1Ø6, no overlength.
   ...
   ...
End Analysis
```

From the informal specification, an input file is read into a buffer of predetermined size. Jackson [2] used this example where the input is read into a buffer by a read_block instruction. He showed how to solve such a problem caused by the differeces between the boundary of bolcks and the boundary of telegrams. His purpose was to show how these problems of "strcuture clashes" in his words, can be solved by introducing an intermediate file and through a program inversion technique. But as he mentioned, there is another solution which is to read the whole input into a buffer or main memory of any finite length.

Concerning the discribing notations, let us use regular expressions and BNF. The operations in regular expressions are " ", "¦", and "*" for concatenation, alternation and repetition including null string, respectively. "+" operation is also used for repetition without null string.

For the regular expressions with repetitive oprations "*" and "+", we can rewrite each rule into some rules without repetitive operations in the following manner.

For the rule with "+" operation where S1 and S2 are any symbols and L and X are nonterminal symbols,

```
<L> ::= <S1> <X>+ <S2>
```

can be transformed into the following two rules:

```
<L>  ::= <S1> <X+> <S2>
<X+> ::= <X> ( <X+> ¦ <null> )
```

For the "*" operation,

```
<L>  ::= <S1> <X>* <S2>
```

is transformed into the followaing two rules:

```
<L>  ::= <S1> <X*> <S2>
<X*> ::= ( <X> <X*> ) ¦ <null>
```

in both cases <null> means an empty string or a string without any symbol.

For the case of the most Prolog implementation, input data is supposed to be a list consisting of character strings. So the input file is to be read into a form of a list of characters, though we do not implement the buffer of predetermined size faithfullly as described in the informal specification. Therefore, we assume the input data will read line by line until the end of file character from the specified file. The data structure of the input file in BNF can be described as in Fig.1. Here <eof> is a special end-of-file character and <cr_lf> is a special new line character (See Appendix).

```
<input_file> ::= <line*>
<line*>      ::= ( <line> <line*> ) ¦ <null>
<line>       ::= <ch*>
<ch*>        ::= ( <ch> <ch*> ) ¦
                 <null>
<ch>         ::= <blank> ¦ <letter> ¦
                 <digit> ¦ <cr_lf>  ¦ <eof>
<blank>      ::= " "
<letter>     ::= "a" ¦ ... ¦ "z" ¦
                 "A" ¦ ... ¦ "Z"
<digit>      ::= "Ø" ¦ ... ¦ "9"
<null>       ::= ""
```

Fig.1 BNF of Input File

4. Derivation of DCG

We will show two programming methods, one of which is useful for a large scale problem, and the other is for a simple problem. The former is based on the BNF for both input and output data structures, introducing an intermediate data structure. Specifically we will use a BNF description of the output report, from which terms and extra conditions in DCG can be derived.

The latter is based on a form of an attribute grammar[10], i.e. a CFG grammar with attributes attached to non-terminal symbols.

4.1 DCG derivation for a large scale program

The following is the overall process to obtain the DCG from BNF.

(1) Represent the syntax rules of the input and output data of a problem in BNF. (Here the left recursive grammar rule is not allowed).

(2) Extract the data in the output which depends on the input data.

(3) For the extracted data in (2), define attributes and arguments related to them (in order to construct a term in Prolog) from the informal problem specification.

(4) Let the intermediate text consist of the data necessary for generating the output data.

(5) Transform the BNFs into DCGs while considering what the arguments of DCG's terms and extra conditions should be.

(6) To execute the input DCG and output DCG under the Prolog environment, prepare the following processes:

(a) Driver process which controls the input DCG, the output DCG and the input process.

(b) Input process which reads the input data from the specified file and generates a list that is one of the basic data structures of Prolog.

By using this method, we can replace the processes for input data analysis and for output data generation by writing DCGs derived from BNFs. Writing an output BNF and a DCG sometimes has another advantage: there is no need to discriminate between parsing and generation. We can generate the output data based on the output BNF as if we parse the output.

In the above procedure, an intermediate text is introduced to link the data for the input and output. The intermediate text will serve as a parse tree obtained from the input, from which the output data is derived. There may be a case where the output data structure is so simple or so similar to the input data struture that we can derive the output data directly from the input, where the parse tree is treated implicitly. On the contrary there may be another case where we must transform the parse tree in order to be more easily used to give an output. We use an intermediate text to show clearly the general procedure in this method. The method discussed in Section 4.2 is also without introducing an intermediate text.

We derive a program of the telegram analysis problem using the method described above.

(1) The data structure of the input data in BNF and the BNF describing the analysis report of the problem can be described as in Fig.2 and Fig.3, respectively.

```
<telegram_stream> ::= <blank*> <telegram*>
                      <empty_telegram>
<blank*>      ::= ( <blank> <blank*> ) | <null>
<telegram*>   ::= ( <telegram> <telegram*> ) |
                  <null>
<null_telegram> ::= <zword>
<telegram>    ::= <non_zword+> <zword>
<non_zword+>  ::= <non_zword> <non_zword+>
<non_zword+>  ::= <non_zword> <null>
<non_zword>   ::= <word> <blank+>
<blank+>      ::= <blank> <blank*>
<zword>       ::= <word> <blank+>
<word>        ::= <char+>
<char+>       ::= <char> <char+>
<char+>       ::= <char> <null>
<char>        ::= <letter> |
                  <digit>
```

Fig.2 BNF of Input Text

```
<report>          ::= <heading_msg>
                      <telegram_msg*>
                      <ending_msg>
<telegram_msg*>   ::= ( <telegram_msg>
                        <telegram_msg*> ) |
                      <null>
<heading_msg>     ::= 'Telegrams Analysis'
                      <new_line>
<telegram_msg>    ::= <tele_id_msg>
                      <charge_msg>
                      <overlength_msg>
                      <new_line>
<ending_msg>      ::= 'End Analysis'
                      <new_line>
<new_line>        ::= <cr_lf>
<tele_id_msg>     ::= 'Telegram '
                      <Telegram_No> '.'
<charge_msg>      ::= 'Charge count is '
                      <Charge_Count> ','
<overlength_msg>  ::= ' overlength occured.' |
                      ' no overlength.'
<Telegram_No>     ::= <integer>
<Charge_Count>    ::= <integer>
```

Fig.3 BNF of Report

(2) We must define other requirements which are not described in these BNFs. According to the problem specification of the report, it is clear that the output report requires the following information for each telegram:

* Telegram number,
* Chargeable Word Count in a telegram and
* Flag of the occurrence of the overlength word in a telegram.

(3) We have to define the meanings of the following three: Telegram Number (or TN), Chargeable Word Count (or CWC) and OverLengthFlag (or OLF), which depend on the input data.

From the requirement of the problem, the chargeable word is neither "ZZZZ" nor "STOP" and the overlength word has a length which is more than twelve characters. Therefore, the following attributes become necessary for Chargeable Word Count and OverLengthFlag:

* chargeable : check a word image and increment Chargeable Word Count if it is neither "ZZZZ" nor "STOP".
* overlength : check a word length and set OverLengthFlag if it is more than twelve.

Each of these attributes is used as a functor of a term which needs the following two data items included in the argument respectively:

* WD_Img : the image which means the character string constructing of a word,
* WD_Len : the length of the word.

To obtain the WD_Img, we use a character string and the following built-in procedure of Prolog:

* name(WD_Img,Ch_Img) : generates an identifier into the variable WD_Img from a list Ch_Img.

For Telegram Number, no special attribute is necessary, and the following term will suffice:

* update_telegram_number : increase the telegram number by one.

(4) Let the intermediate text, IT, consist of data depending on the input data which has the list structure of the triple used for generating the output data. That can be represented in Prolog notation as follows:

```
[ [TN1,CWC1,OLF1], ... ,[TNn,CWCn,OLFn] ]
```

(5) We use the DCG to add the necessary attributes to the specifications of the input text and the output report described in BNF.

We must determine those arguments, only which can be used for passing information, in each term. In our case, the information which is to be obtained as a result is three data items, TN, CWC and OLF, as described above. They should be computed from an initial value through intermediate values, until the result is obtained. Due to the characteristic of the Prolog which is one assignement rule for a variable, we will use arguments with labels prefixed with Old_ and New_ for data passing from the source to the target. For example, Old_CWC and New_CWC are used for the CWC.

Now each of the information from the specification can be embedded into DCG as follows:

(a) A data item is used as an argument.
(b) An attribute is used to construct a term, which is embedded in an extra condition.

The attribute update_telegram_number needs TN and New_TN. The following is the definition:

```
update_telegram_number(TN,New_TN) :-
                              New_TN is TN + 1.
```

The built-in procedure,"is", of Prolog means that the left-hand side variable are instantiated by an evaluated value of the right-hand side expression.

The attribute "chargeable" needs WD_Img, Old_CWC and New_CWC as arguments. And also the attribute "overlength" needs WD_Len, Old_OLF and New_OLF as arguments.

The followings are the definition:

```
chargeable(WD_Img,Old_CWC,New_CWC) :-
        ( WD_Img \== 'ZZZZ',
          WD_Img \== 'STOP',
              New_CWC is Old_CWC + 1 ) ;
              New_CWC is Old_CWC.
overlength(WD_Len,Old_OLF,New_OLF) :-
        ( WD_Len > 12, New_OLF = true ) ;
                       New_OLF = Old_OLF.
```

Fig.4 shows the DCG form of the input text which includes the attributes of the telegrams. The definitions of attributes are described as Prolog procedures. We use arguments to pass the information between non-terminal symbols. When the argument needs the variable information, we use the concept of the logical variable in Prolog (which begins with a capital letter).

Fig.5 shows the DCG form of the reporting. We use the write function as an extra condition instead of generating the list of the report.

```
telegram_stream(IT) -->
     blank_star, telegram_star(1,IT),
     empty_telegram.
blank_star --> ( blank, blank_star ) ;
               null.
telegram_star(TN,[[TN,CWC,OLF]|Ts]) -->
     telegram(TN,CWC,OLF),
     { update_telegram_number(TN,New_TN) },
     telegram_star(New_TN,Ts).
telegram_star(TN,[]) --> null.
empty_telegram --> zword.
telegram(TN,CWC,OLF) -->
     non_zword_pls(CWC,OLF), zword.
non_zword_pls(New_CWC,New_OLF) -->
     non_zword(WD_Img,WD_Len),
     non_zword_pls(Old_CWC,Old_OLF),
     { chargeable(WD_Img,Old_CWC,New_CWC),
       overlength(WD_Len,Old_OLF,New_OLF) }.
non_zword_pls(New_CWC,New_OLF) -->
     non_zword(WD_Img,WD_Len), null,
     { chargeable(WD_Img,0,New_CWC),
       overlength(WD_Len,false,New_OLF) }.
non_zword(WD_Img,WD_Len) -->
     word(WD_Img,WD_Len), blank_pls,
     { WD_Img \== 'ZZZZ', WD_Len > 0 }.
word(WD_Img,WD_Len) -->
     char_pls(Ch_Img,Ch_Len),
     { name(WD_Img,Ch_Img), W_Len = Ch_Len }.
zword --> word(WD_Img,WD_Len), blank_pls,
          { WD_Img = 'ZZZZ', WD_Len = 4 }.
blank_pls --> blank, blank_star.
char_pls(Ch_Img,Ch_Len) -->
     char(C), char_pls(Sub_Img,Sub_Len),
     { Ch_Img = [C|Sub_Img],
       Ch_Len is 1 + Sub_Len }.
char_pls(Ch_Img,Ch_Len) --> char(C), null,
     { Ch_Img = [C], Ch_Len is 1 }.
char(C) --> letter(C) ;
            digit(C).
```

Fig.4 The DCG Form of Input Analysis of Telegram

```
report(IT) --> heading_msg,
               telegram_msg_star(IT),
               ending_msg.
telegram_msg_star([X|Y]) -->
     telegram_msg(X), telegram_msg_star(Y).
telegram_msg_star([])    --> null.
heading_msg -->{ write('Telegrams Analysis') },
               new_line.
telegram_msg([TN,CWC,OLF]) -->
     tele_id_msg(TN), charge_msg(CWC),
     overlength_msg(OLF), new_line.
ending_msg --> { write('End Analysis') },
               new_line.
new_line --> { nl }.
tele_id_msg(TN) --> { write('Telegram '),
                      write(TN), write('.') }.
charge_msg(CWC) -->
     { write(' Charge count is '), write(CWC),
       write('.') }.
overlength_msg(OLF) -->
     {(OLF=true,write(' overlength occured.'));
      (OLF=false,write(' no overlength.') ) }.
```

Fig.5 The DCG Form of Telegram Reporting

(6) Fig.6 shows the Prolog program for the main control of the telegram analysis. Fig.7 shows the input file program for the input data from the specific file. This program can be derived from BNF of Input File, Fig.1. To obtain characters from the specified file, we use the following built-in procedures of Prolog.

* get0(X) : gets the next character from the current input stream into the variable X.
* see(X) : opens file X as the current input stream.
* seen : closes the current input stream.

```
telegram_analysis(File_Name) :-
    input_file(File_Name,Character_String,[]),
    input_text(IT,Character_String,[]),
    report(IT,Temp,[]).
```

Fig.6 Prolog Form of Driver Process

```
input_file(File_Name) -->
     {see(File_Name)}, line_star(EOF), {seen}.
line_star(EOF) -->
     ( { EOF \== eof },
       line(EOF), line_star(EOF) ) ;
     null.
line(EOF) --> ch_star(CR_LF,EOF).
ch_star(CR_LF,EOF) -->
     ( { CR_LF \== on, EOF \== on },
       ch(CR_LF,EOF), ch_star(CR_LF,EOF) ) ;
     null.
ch(CR_LF,EOF) -->
     { get0(X) },                          /* =>*1 */
     ( ( { X = 32 }, blank )              ;/* =>*2 */
       letter( X )                        ;
       digit( X )                         ;
       ( { X = 31 }, cr_lf(CR_LF) )       ;/* =>*3 */
       ( { X = 26 }, eof(EOF) ) ).         /* =>*4 */
blank     --> " ".
letter(C) --> [C],{(( "a" =< C, C =< "z" ) ;
                    ( "A" =< C, C =< "Z" )  )}.
digit(C)  --> [C], { "0" =< C, C =< "9" }.
cr_lf(CR_LF) --> { CR_LF = on }.
eof(EOF)     --> { EOF = on }
null         --> "".
```

Fig. 7 DCG Form of Input File

Remark *1 : gets a character into X from the file specified File_Name.
*2 : X = 32 means that X is ASCII blank.
*3 : X = 31 means that X is a special end of line character.
*4 : X = 26 means that X is a special end of file character.

Two extra arguments are added to input_file, input_text and report. They are necessary for a Prolog term which communicates with a DCG term. One argument is analysed and the other is the termination list for terminating the analysis of the input data. The input_text(IT,Character_String,[]) analyses the list in the variable "Character_String" until arriving at the empty list, "[]", and generates the result into the variable IT. The report(IT,Temp,[]) uses the list in the variable IT and generates the output data into the variable "Temp".

4.2 DCG derivation for a simple problem

Someone may consider the method described in Section 4.1 too complicated for deriving the telegram analysis program. There may be no needs to introduce an intermidiate data structure where the output data structure is so simple or so similar to the input data structure, since the output data can be derived directly from the input. Thus in this Section we will show a simple and straightforward derivation method from the input data structure directly.

In [6] an attribute grammar for the telegram analysis problem has been shown. He tryied to derive a procedural program like PL/I, although it is not easy to derive a program from the attribute grammar.

Let us define two kinds of attributes for each BNF in the following manner:

```
LHS -> RHS
  [when]:{C}
  [result]:{R}
```

LHS in a non-terminal symbol and RHS is a term of regualer expressions, of the form of CFG's rewriting rules. C, called "conditional attribute", is a set of conditions, i.e. when all of the elements of the set C are satisfied, the rule can be applied. The other attribute R called "resulting attribute", is a set of actions, i.e. after the rule is applied, all of the actions in this set should be done.

For example, the input is a stream of telegrams which can be written as

```
<telegram_stream> ::=
          <blank>* <telegram>* <null_telegram>
     [result]: {heading_msg and ending_msg}
```

The attributed version of BNF is showed in Fig.8. The attributes such as TN, CWC, OLF, WD_Img, WD_Len, Ch_Img, Ch_Len are the same ones used in Section 4.1. So we don't discuss them here. The resulting attributes, heading_msg, ending_msg and telegram_msg, are almost same as ones used in Fig.3. So we don't discuss them too.

```
<telegram_stream> ::=
          <blank*> <telegram*> <null_telegram>
     [result]: { heading_msg and ending_msg }
<blank*> ::= ( <blank> <blank*> ) |
             <null>
<telegram*> ::= ( <telegram> <telegram*> ) |
                <null>
<null_telegram> ::= <zword>
<telegram> ::= <non_zword+> <zword>
     [result]:{ telegram_msg([TN,CWC,OLF]) }
<non_zword+> ::= <non_zword> <non_zword+>
     [when]:{ WD_Img \== 'ZZZZ' }
     [result]:{ if WD_Img \== 'STOP'
                   then New_CWC := Old_CWC + 1
                   else New_CWC := Old_CWC
                if WD_Len > 12
                   then New_OLF := true
                   else New_OLF := Old_OLF }
<non_zword+> ::= <non_zword> <null>
     [when]:{ WD_Img \== 'ZZZZ' }
     [result]:{if WD_Img  \==  'STOP'
                   then New_CWC := 1
                   else New_CWC := Ø
              if WD_Len  > 12
                   then New_OLF := true
                   else New_OLF := false }
<non_zword> ::= <word> <blank+>
     [when]:{WD_Img \== 'ZZZZ' and WD_Len > Ø}
<blank+> ::= <blank> <blank*>
<zword> ::= <word> <blank+>
     [when]:{ WD_Img = 'ZZZZ' and WD_Len = 4 }
<word> ::= <char+>
     [result]:{ WD_Img := Ch_Img and
                WD_Len := Ch_Len }
<char+> ::= <char> <char+>
     [result]:{ Ch_Img := C & Sub_Img and
                Ch_Len := 1 + Sub_Len }
<char+> ::= <char> <null>
     [result]:{ Ch_Img := C and Ch_Len := 1 }
<char> ::= <letter> |
           <digit>
```

Fig.8 BNF with attributes for Telegram Stream

Fig.9 shows the DCG Form and the Prolog program derived from BNF in Fig.8.

```
telegram_stream --> { heading_msg },
     blank_star, telegram_star(1),
     empty_telegram,
     { ending_msg }.
blank_star --> ( blank, blank_star ) ;
               null.
telegram_star(TN) -->
      ( telegram(TN), { New_TN is TN + 1 },
        telegram_star(New_TN) ) ;
       null.
empty_telegram --> zword.
telegram(TN) --> non_zword_pls(CWC,OLF), zword,
                 { telegram_msg(TN,CWC,OLF) }.
non_zword_pls(New_CWC,New_OLF) -->
     non_zword(WD_Img,WD_Len),
     non_zword_pls(Old_CWC,Old_OLF),
     { WD_Img \== 'ZZZZ',
       (( WD_Img \== 'STOP',
                  New_CWC is Old_CWC + 1 ) ;
                  New_CWC is Old_CWC        ),
       (( WD_Len > 12, New_OLF = true ) ;
                       New_OLF = Old_OLF  ) }.
non_zword_pls(New_CWC,New_OLF) -->
     non_zword(WD_Img,WD_Len), null,
     { WD_Img \== 'ZZZZ',
       (( WD_Img \== 'STOP', New_CWC is 1 ) ;
                             New_CWC is 0  ),
       (( WD_Len > 12, New_OLF = true ) ;
                       New_OLF = false   ) }.
non_zword(WD_Img,WD_Len) -->
     word(WD_Img,WD_Len), blank_pls,
     { WD_Img \== 'ZZZZ', WD_Len > 0 }.
blank_pls --> blank, blank_star.
word(WD_Img,WD_Len) -->
     char_pls(Ch_Img,Ch_Len),
     { name(WD_Img,Ch_Img), WD_Len = Ch_Len }.
char_pls(Ch_Img,Ch_Len) -->
     char(C), char_pls(Sub_Img,Sub_Len),
     { Ch_Img = [C|Sub_Img],
       Ch_Len is 1 + Sub_Len }.
char_pls(Ch_Img,Ch_Len) --> char(C), null,
     { Ch_Img = [C], Ch_Len is 1 }.
zword --> word(W_Img,W_Len), blank_pls,
          { W_Img = 'ZZZZ' }.
char(C) --> letter(C) ;
            digit(C).
heading_msg :- write('Telegrams Analysis'), nl.
telegram_msg(TN,CWC,OLF) :-
     tele_id_msg(TN), charge_msg(CWC),
     overlength_msg(OLF), nl.
tele_id_msg(TN) :- write('Telegram '),
                   write(TN), write('.').
charge_msg(CWC) :- write(' Charge count is '),
                   write(CWC), write('.').
overlength_msg(OLF) :-
     (OLF=true, write(' overlength occured.'));
     (OLF=false, write(' no overlength.') ).
ending_msg :- write('End Analysis'), nl.
telegram_analysis(File_Name) :-
     input_file(File_name,Char_String,[]),
     telegram_stream(Char_String,[]).
```

Fig.9 DCG and Prolog Form of Telegram Analysis

4. Conclusions

We have described, in this paper, two rigorous methods for obtaining correct programs. These two methods are to derive DCG grammar from BNF rules of input/output data structures. The first introduces an intermediate file which is useful for the case when the input and output data structures are not similar. We have applied this method to a file manipulation problem without difficulty. We have obtained a hint as to the necessity or convenience of introducing an intermediate file from Jackson [2]. His method to solve the structure clashes can be used straightforwardly. Therefore, the applicable domain of this method will at least be the domain of Jackson's book. The second method is simpler than the first, since it does not use an intermediate file. This method is usefull when there is no substantial differences between the input/output data structures.

Each method of using the Definite Clause Grammar in Prolog for text processing problems, as discussed in Section 4, is one of the easiest ways for rapid prototyping. Because if the input data structures changes, it is enough to rewrite the input BNF rules. And if the output data structure changes or more sophisticated process is needed, rewriting the output BNF rules or the attributes for rules can accomplish the object of changes. If we can develop a faster Prolog processor in terms of speed of execution, this approach should be practical. In order to increase the speed of execution "Cut" operation can be used as in the usual Prolog program, which restricts the automatic backtracking.

What we plan to improve from now on are the removal of left recursive constraints coming from Prolog's top down parsing by modifying the Prolog processors, and an automatic translation in the second method from BNF with attributes to Prolog programs.

References

1. Henderson,P. and Snowdon,R., An Experiment in Structured Programming, BIT, Vol.12, 38-53 (1972).
2. Jackson,M.A., Principles of Programs Design, Academic Press (1975).
3. Jones,C.B., Software Development : A Rigorous Approach, Prentice-Hall, 325-332 (1980).
4. Ledgard,H.F., The Case for Structured Programming, BIT, Vol.12, 45-57 (1973).
5. Mckeag,R.M. and Milligan,P., An Experiment in Parallel Program Design, Software-Practice and Experience, Vol.10, 687-696 (1980).
6. Noonan,R.E., Structured Programming and Formal Specification, IEEE Trans. on Software Engineering, Vol.SE-1, No.4, 421-423(1975).
7. Pereira,L., Pereira,F. and Warren,D., User's Guide to DECsystem-10 Prolog, Div. de Infomatica, LNEC, Lisbon and Dept. of AI, University of Edinburgh(1978).

8. Pereira,F. and Warren,D., Definite Clause Grammars for Language Analysis, Artificial Intelligence 13, 231-278(1980).
9. Torii,K., Morisawa,Y., Sugiyama,Y. and Kasami,T., A Functional Programming and Logical Programming for the Telegram Analysis Problem, Proceeding of the 8th ICSE, 463-472(1984).
10. Knuth,D.E., Semantics of Context-Free Languages, Math. Syst. Th., Vol.2, No.2, 127-145(1968).

Appendix:

When we implement the buffer of predetermined size fathfully as described in the informal specification, Fig.1 and Fig.7 are replaced by Fig.10 and Fig.11 respectively. The program in Fig.11 assumes the buffer size is 80 characters.

```
<input_file> ::= <block*>
<block*>     ::= ( <block> <block*> ) |
                 <null>
<block>      ::= <ch_in_blk*>
<ch_in_blk*> ::= ( <ch_in_blk> <ch_in_blk*> ) |
                 <null>
<ch_in_blk>  ::= <blank> | <letter> | <digit> |
                 <cr_lf> | <eof>
<blank>      ::= " "
<letter>     ::= "a" | ... | "z" |
                 "A" | ... | "Z"
<digit>      ::= "0" | ... | "9"
<null>       ::= ""
```

Fig.10 BNF of Input File

```
input_file(Fname) -->
     {see(Fname)}, block_star(80,EOF), {seen}.
block_star(BS,EOF) -->
     ( { EOF \== eof },
       block(BS,EOF), block_star(BS,EOF) ) ;
     null.
block(BS,EOF) --> ch_in_blk_star(1,BS,EOF).
ch_in_blk_star(Pointer,BS,EOF) -->
     { EOF \== eof , Pointer =< BS },
     ch_in_blk(EOF),
     { New_Pointer is Pointer + 1 },
     ch_in_blk_star(New_Pointer,BS,EOF).
ch_in_blk_star(Pointer,BS,EOF) -->
     { ( EOF \== eof, Pointer > BS ) ;
       ( EOF  =  eof, Pointer < BS )  }.
ch_in_blk(EOF) -->
     { EOF \== eof, get0(X) },
     ( ( { X = 32 }, blank )              ;
       letter( X )                        ;
       digit( X )                         ;
       ( { X = 31 }, ch_in_block(EOF) ) ;
       ( { EOF = eof}, blank )             ).
blank     --> " ".
letter(C) --> [C],{(( "a" =< C, C =< "z" ) ;
                    ( "A" =< C, C =< "Z" )  )}.
digit(C)  --> [C], { "0" =< C, C =< "9" }.
null      --> "".
```

Fig.11 DCG Form of Input File