ISE-TR-88-73

Programming in Gramp:

a programming language based on CCFG

by

Yoshiyuki YAMASHITA and Ikuo NAKATA

June 1, 1988

INSTITUTE
OF
INFORMATION SCIENCES AND ELECTRONICS

UNIVERSITY OF TSUKUBA

Programming in Gramp:
a programming language based on CCFG

Yoshiyuki YAMASHITA, *the Doctoral Program in Engineering,*
and
Ikuo NAKATA , *the Institute of Information Sciences and Electronics,*
University of Tsukuba,
Tsukuba-shi, Ibaraki-ken 305.

Abstract

Gramp is a declarative programming language based on Coupled Context-Free Grammar (CCFG). It is considered that Gramp is effective to express the programs whose input/output data objects have some kind of structures, especially hierarchical ones. In this sense CCFG formalizes and Gramp embodies the Jackson program design. Here some problems described in [1] are expressed in Gramp programs. It is shown that the programs are more readable than in other languages. Several ideas of syntax-sugaring and control in Gramp are also discussed.

[1] Jackson, M. A. : *Principles of Program Design,* Academic Press (1975).

1.Introduction

CCFG programming [2] is a programming paradigm in which a context-free grammar is regarded as the representation of an input/output data structure and a couple of the context-free grammars, called a Coupled Context-Free Grammar (CCFG) or a CCFG program, is regarded as a program representing the relation between such data structures. It has been proved that CCFG programs with some additional devices can express every recursively enumerable set [2]. Therefore we say that CCFG programming is universally descriptive. For example, the following is a CCFG program representing the relation of string reversal.

$$\{X\rightarrow\varepsilon, \quad Y\rightarrow\varepsilon\}$$
$$\{X\rightarrow aX, \quad Y\rightarrow Ya\}$$
$$\{X\rightarrow bX, \quad Y\rightarrow Yb\}$$

Here ε is an empty symbol, $A\rightarrow\alpha$ is a production rule, and each set of production rules enclosed by braces is called a rule-set, in which all the production rules must be simultaneously applied. Therefore one of the derivation from the couple (X, Y) of start symbols is given as follows,

$$(X, Y)\Rightarrow(aX, Ya)\Rightarrow(abX, Yba)\Rightarrow(abbX, Ybba)\Rightarrow(abb, bba).$$

In this way, we see that every couple of terminal strings derived from (X,Y) has two strings which are mutually reversed. This is the reason we say that this CCFG program represents the string

reversal.

In the sense that a CCFG represents a relation between input/output data objects, it is quite similar to a logic program. We have obtained the program transformation rules from a logic program into the equivalent CCFG program and vice versa. These transformation rules are simple enough to automate the transformation. This relationship between two equivalent logic and CCFG programs shows that the semantic structures of both programs are similar and their syntactic structures are just the reverse of each other. Namely both programs are in the dual relation. Since the readability of a program is greatly influenced by the syntactic structure of the program, problems suitable for solving in logic programming are not always suitable for CCFG programming and vice versa. The main purpose of this paper is to clarify the domain of problems which are suitable for solving in CCFG programming rather than logic programming.

The Jackson program development method uses the diagrams which are equivalent to regular expressions for representing input/output data structures at the first stage of the development. A set of regular expressions is equivalent to a context-free grammars. In this sense the Jackson method is considered to be closely related to CCFG programming. In fact, one of our motivations to invent CCFG programming was to formalize the Jackson method in a declarative programming paradigm. Therefore some of the problems discussed in [1] are expected to be compactly and straightforwardly expressed in CCFG programs rather than in other style of programs, especially in COBOL. In this paper we show this by solving three examples in [1].

Gramp (a **Gramm**ar is a **p**rogram !) is a programming language based on CCFG and designed for practical use [4]. It can treat character strings, terms, rational numbers and their mixtures. From the unified theory of CCFG programs with logic and functional ones, Gramp can also treat predicate-calls and function-calls [4]. In this paper, in addition to such basic features we introduce some words for syntax-sugaring and control structures to Gramp, which make Gramp programs more readable.

In section two, we overview Gramp by using a small program.

In section three, we solve the problem of counting batches [problem 4 in [1] pp.49-50]. In this problem the data structure of the input sequence of cards determines the whole program structure. The usage of regular expressions and an iterative control structure are introduced. The program transformation by loop fusion is also discussed.

In section four, we solve the problem of the Magic Mailing Company [problem 7 in [1] pp.70-71]. The syntax-sugaring words `"where"`, `"let"` are used in order to make the meanings of rule-sets obvious. The word `"otherwise"` is also introduced, which give priority to rule-sets selectively. This problem is solved in both bottom-up and top-down manners.

In section five, we solve the problem of telegrams analysis [problem 13 in [1] pp.155-156]. It is known that this problem can not be solved without an intermediate file. In Gramp such a file is implemented by connecting two nonterminal symbols with an equality symbol == which is the same as the metasymbol ≈ in CCFG [2]. The word `"with"` is also introduced.

In section six, we compare a Gramp program with a Prolog program [7]. We will see that the problems whose input/output data structures are complex and hierarchical are suitable for solving in Gramp because such data structures can be easily defined by context-free grammars and the definition is more readable than that defined in clausal forms. But the problems which have many

complicated relations are suitable for solving in Prolog because logical connectivity among relations (literals) is important for their program structures rather than input/output data structures.

2.The programming language: Gramp

The programming language: Gramp is explained here by showing a small program.

Character strings are the basic units in Gramp, which are explicitly written by enclosing the sequence of characters by double quotes. The empty string is denoted by two consecutive double quotes "", and the concatenation operator is a colon :. The following conditions hold for arbitrary strings "α", "β" and "γ".

("α":"β"):"γ" = "α":("β":"γ") = "αβγ",

"":"α" = "α":"" = "α".

The subprograms in Gramp are **modules**. For example, a module representing the string reversal is written as follows. We have already seen the same CCFG program in the previous section.

```
module rev(Input,Output)
{ Input->"",          Output->"" }                    ...(rev.1)
{ Input->"a":Input,  Output->Output:"a" }            ...(rev.2)
{ Input->"b":Input,  Output->Output:"b" }            ...(rev.3)
end
```

Here `rev` is a module name, which must begins with lower case letters. The symbols `Input` and `Output` are **nonterminal symbols**, which must begin with upper case letters. The symbols `Input` and `Output` are called **input/output symbols** because they are declared as `rev(Input,Output)` at the head of the module. An input/output symbol is an interface of the module with other modules and can be referred from the outside of the module, while a non-input/output symbol cannot. The body of the module is a set of the **rule-sets**. A rule-set is a set, enclosed by braces, of **context-free production rules** of the form `A->`α where the left-hand side `A` is a nonterminal symbol. Any two left-hand sides in a rule-set must not be the same.

Since all the production rules in a rule-set are to be *simultaneously* applied [2], a derivation sequence from the tuple `(Input, Output)` of input/output symbols is, for example, given as follows,

```
(Input, Output)⇒(  "a":Input, Output:"a")          by (rev.2)
               ⇒( "ab":Input, Output:"ba")         by (rev.3)
               ⇒("abb":Input, Output:"bba")        by (rev.3)
               ⇒("abb",              "bba").       by (rev.1)
```

The derived string "abb" is the reversal of "bba" and vice versa. In the same way as above, we see that any two strings derived are reversed in general. Therefore the module `rev` can be said to be a module which represents the string reversal. To speak more formally, the semantics *SF*[`rev`] of the module is denoted by the set of all the couples of character strings derived from the module as follows,

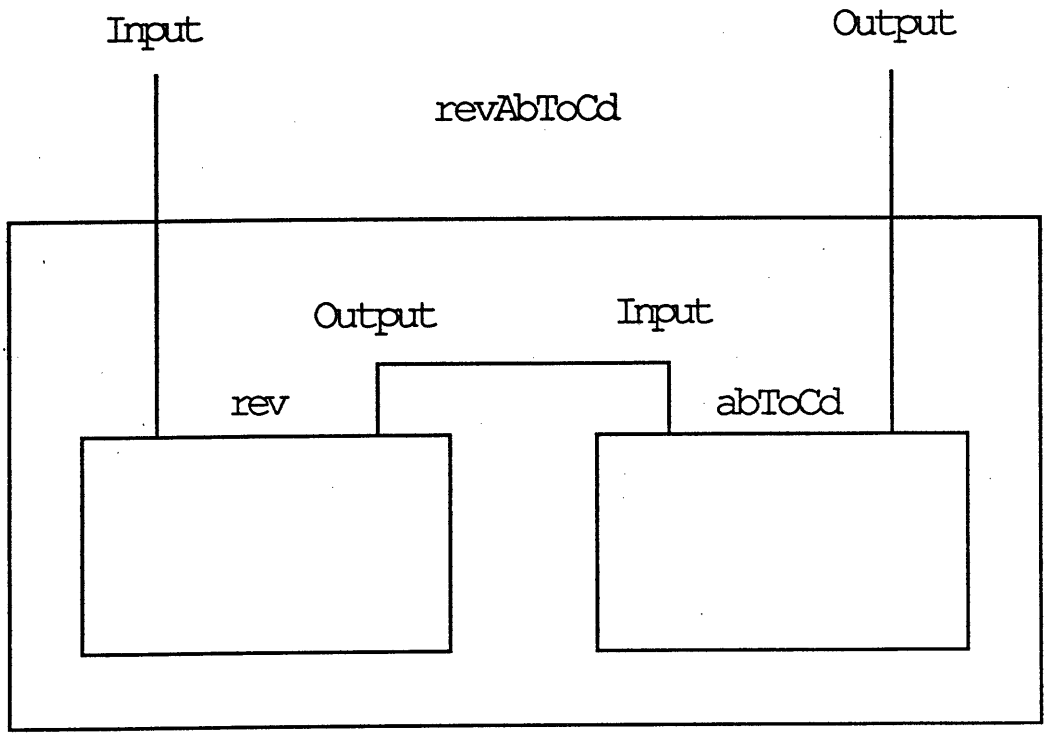

Figure 1. The structure of the compound module revAbToCd

SF[rev] = {("", ""), ("a", "a"), ("b", "b"), ("ab", "ba"), ("ba", "ab"), ...}.

Remarks Although we use the symbol name "Input" and "Output" for convenience, the module does not express an one way transformation like a scheme of syntax directed translations [6]. It expresses a relation between Input and Output. Therefore we can transform a string not only from Input to Output, but also from Output to Input.

The following is the module to transform the strings composed of the characters "a" and "b" into the strings of "c" and "d".

```
module abToCd(Input,Output)
{ Input->"",          Output->"" }
{ Input->"a":Input,  Output->"c":Output }
{ Input->"b":Input,  Output->"d":Output }
end
```

The semantics of this module can be obtained in the same way as in the previous case.

The above two modules can be connected by the following compound module, which transforms the strings composed of "a" and "b" into the reversed strings of "c" and "d".

```
module revAbToCd(Input,Output)
{ Input->rev.Input, rev.Output==abToCd.Input, Output->abToCd.Output }
end
```

This module has only one rule-set. Here the symbol == is equivalent to *the* metasymbol ≈ in CCFG programs (see [2], [3]), whose intuitive meaning is the same as that of the equality symbol in

equational logic. The symbols `rev.Input` and `rev.Output` in the above rule-set means `Input` and `Output` in the module `rev`. Therefore `rev.Input` and `rev.Output` are simultaneously rewritten and they derive the mutually reversed character strings. This means a **module invocation** in Gramp. In the same way, `abToCd.Input` and `abToCd.Output` express the invocation of the module `abToCd`. The string `"rev.Output==abToCd.Input"` has no left-hand side nonterminal symbol followed by ->. This is called a **left-hand-omitted-rule** (see [2]) and it means that the two symbols `rev.Output` and `abToCd.Input` must always derive a same character string. Therefore this left-hand-omitted-rule expresses the communication between the modules `rev` and `abToCd` through the channel == in Gramp. The module `revAbToCd` is a compound module (see figure 1).

A **main program** in Gramp contains only one rule-set which is composed of only left-hand-omitted-rules in the similar sense that a main program in a logic program (pure Prolog) is a Horn clause which has no literal in its head. For example, the following is a main program whose name is `"smallProgram"`.

```
main smallProgram
{ "abb"==revAbToCd.Input, revAbToCd.Output==?SysOut }
end
```

This program is to obtain the string corresponding to the input string `"abb"` and to send it to the system's output device named `"?SysOut"`. In this paper we do not discuss the functions and actions of such physical devices in Gramp. If we do not use the compound module, the program may be rewritten as follows.

```
main smallProgram2
{ "abb"==rev.Input, rev.Output==abToCd.Input, abToCd.Output==?SysOut }
end
```

And further if we do not use module invocations, it may also be rewritten as follows.

```
main smallProgram3
{ "abb"==Input1, Output1==Input2, Output2==?SysOut }

{ Input1->"",          Output1->"" }
{ Input1->"a":Input1, Output1->Output1:"a" }
{ Input1->"b":Input1, Output1->Output1:"b" }

{ Input2->"",          Output2->"" }
{ Input2->"a":Input2, Output2->"c":Output2 }
{ Input2->"b":Input2, Output2->"d":Output2 }
end
```

Though we have treated only character strings in the above example, we can also treat terms (which include lists) and rational numbers as primitive data types in Gramp. There is no theoretical difficulty to treat them (such theoretical discussion will be given in separate paper). The Gramp program in this section is quite small. In the following sections we try to build more complicated Gramp programs.

3.Counting batches

Now we solve the problem 4 in [1]. In subsection 3.1, we obtain a naive Gramp program, and in subsection 3.2, 3.3 and 3.4, we revise it and obtain more effective one.

Problem 4 An input file of card images is to be analyzed. There are three card types, T1, T2 and T3, with the values 1, 2 and 3 respectively in position 1 of the card. The required analysis is as follows:

(1) Count the cards preceding the first T1 (count A).
(2) Display the first T1.
(3) Display the last card, which is always the first T2 following the first T1.
(4) Count the batches following the first T1, where a batch is either an uninterrupted succession of one or more T1 cards or an uninterrupted succession of one or more T3 cards (count B).
(5) Count the T1 cards after the first T1 card (count C).
(6) Count the batches following the first T1 card which consist of T3 cards (count D).

All counts are to be displayed following the display of the last card. The file is known to be in correct format; that is, there is at least one T1 card, the last card is a T2, and no T2 intervenes between the first T1 and the last card.

3.1.Naive program

The following singleton sets are rule-sets which define the formats of the card images T1, T2 and T3.

```
{ T1->"1":CharString:\cr\ }
{ T2->"2":CharString:\cr\ }
{ T3->"3":CharString:\cr\ }
```

These rule-sets mean that the nonterminal symbols `T1`, `T2` and `T3`, which express the card images, are character strings which begin with the character `"1"`, `"2"` and `"3"` respectively, and end with the carriage-return `\cr\`. The system-defined nonterminal symbol `CharString` expresses an arbitrary character string.

Next we obtain the subprograms which count A, B, C and D according to the requirements (1), (4), (5) and (6) in the above problem, respectively. The subprogram to count A is given as follows,

```
{ InputA->"",              A->0 }
{ InputA->T2orT3:InputA,  A->1+A }
```

where `InputA` expresses the sequence of cards preceding the first `T1`, and `A` expresses the corresponding count. The first rule-set means that if `InputA` is empty, the corresponding `A` is zero. The second means that if `InputA` is composed of `T2orT3` and `InputA`, the corresponding `A` is `1+A`. This is a recursive definition of the relation between `InputA` and `A`. The nonterminal symbol `T2orT3` is defined as follows,

```
{ T2orT3->T2 }
{ T2orT3->T3 }
```

Namely T2orT3 is either T2 or T3. Hence InputA is defined as an arbitrary string composed of T2 and T3. One example of derivations from the couple (InputA,A) is given as follows,

```
(InputA, A) ⇒ (T2orT3:InputA,          1+A)
            ⇒ (T2:InputA,              1+A)
            ⇒ (T2:T2orT3:InputA,       2+A)
            ⇒ (T2:T3:InputA,           2+A)
            ⇒ (T2:T3:T2orT3:InputA,    3+A)
            ⇒ (T2:T3:T2:InputA,        3+A)
            ⇒ (T2:T3:T2,               3)
```

This means that if the input string InputA is the sequence of cards T2, T3 and T2 in this order, the corresponding count A, which is the number of cards in the InputA, is three. In this way, the above rule-sets satisfy the requirement (1).

In the same way, the subprograms to count B, C and D are given as follows,

```
{ InputB->InputB1,  B->B1 }
{ InputB->InputB3,  B->B3 }
    { InputB1->"",            B1->0 }
    { InputB1->T1s:InputB3,   B1->1+B3 }
    { InputB3->"",            B3->0 }
    { InputB3->T3s:InputB1,   B3->1+B1 }

{ InputC->"",         C->0 }
{ InputC->T1:InputC,  C->C+1 }
{ InputC->T3:InputC,  C->C }

{ InputD->InputD1, D->D1 }
{ InputD->InputD3, D->D3 }
    { InputD1->"",            D1->0 }
    { InputD1->T1s:InputD3,   D1->D3 }
    { InputD3->"",            D3->0 }
    { InputD3->T3s:InputD1,   D3->1+D1 }
```

Here T1s and T3s are defined as follows,

```
{ T1s->T1 }
{ T1s->T1:T1s }
{ T3s->T3 }
{ T3s->T3:T3s }.
```

The subprogram to count B and D are a little more complicated than those to count A and C, because the former subprograms have to recognize *uninterrupted successions* of one or more cards of the same type. In the case of the subprogram to count B, by using two distinct pairs (InputB1,B1) and (InputB3,B3) of nonterminal symbols, two states of the recognizer and the state transitions from one to the other are implemented. In the state of InputB1 the subprogram reads only T1

cards, and in the InputB3 state only T3 cards. Counting-up of B1 (or B3) is invoked when a state transition arises. More elegant program which does not use the notion of states is given in subsection 3.3.

Now we give the whole structure of our program as follows,

```
{ CardSeq->InputA:T1:(InputB==InputC==InputD):T2,
  Output->T1:T2:
          "A=":!charStr(A):\cr\:
          "B=":!charStr(B):\cr\:
          "C=":!charStr(C):\cr\:
          "D=":!charStr(D):\cr\ }
```

The first rule means that CardSeq consists of the sequence InputA of T2 and T3 preceding the first T1, the sequence InputB==InputC==InputD of T1 and T3, and the last card T2. The sequence InputB==InputC==InputD means that one sequence is interpreted in multiple ways as InputB, InputC, and also InputD. In this sense it is said that CardSeq has a **multiple data structure**. It is easy to understand that this production rule satisfies the requirement of the problem. The second rule means that Output consists of the first T1, the last T2, and the values of the counts A, B, C, and D which correspond to InputA, InputB, InputC, and InputD, respectively, where !charStr(X) is a function call which returns the character string corresponding to the value of X.

At the end, we connect the symbol CardSeq with a system input file named "inputFile" and the symbol Output with a line printer. The following is the main program to solve the problem 4,

```
main problem4
{ ?Input("inputFile")==CardSeq, Output==?LinePrinter }
```

followed by all of the above rule-sets, where ?Input(...) and ?LinePrinter are the special nonterminal symbols which express a file input device and a line printer output device, respectively. In this paper we do not discuss the actions of such devices.

3.2.Regular expressions

The production rule whose right-hand side is a regular expression is more expressive and more readable than the corresponding set of usual context-free rules. In Gramp regular expressions can be used as the right-hand sides of production rules. For example, T2orT3, T1s and T3s are defined as follows,

```
{ T2orT3->T2|T3 }
{ T1s->T1:{T1} }
{ T3s->T3:{T3} }
```

Here "α|β" means the alternative of α or β and "{α}" the 0 or more repeated α. By using these expressions, simple data structures can be expressed more concisely.

If a rule-set has two context-free rules of which some right-hand sides have the similar structure of regular expressions as follows,

```
{ X->a:(b|c), Y->(d|e):f, Z->g, ... }
```

it is considered to be equivalent to the following rule-sets,

```
{ X->a:b, Y->d:f, Z->g, … }
{ X->a:c, Y->e:f, Z->g, … }.
```

And the following rule-set

```
{ X->a:{b}, Y->{c}:d, Z->e, … }
```

is equivalent to the following rule-sets,

```
{ X->a,   Y->d,   Z->e, … }
{ X->X:b, Y->c:Y, Z->Z, … }.
```

A rule-set must not contain any two context-free rules whose right-hand sides have different structures of regular expressions. For example, the following rule-set is not allowed in Gramp.

```
{ X->a:(b|c), Y->{d}:f, Z->g, … }
```

because the structure of `a:(b|c)` is different from that of `{d}:f`. Regular expressions should be used for making the expressions of (rather) simple data structures more concise.

Note The context-free grammars in which the right-hand side of every production rule is allowed to be a regular expression over terminal and nonterminal symbols, are called regular right part grammars. They have been studied well in [9]. Our discussions here are based on them.

3.3.Expression of uninterrupted successions

In the above program, we have used the notion of state transitions when writing the subprograms to count B and D. They have the two states only. If a program has more states, however, it is obvious that the program becomes more difficult to read. The essence of subprograms to count B and D is to recognize the uninterrupted successions of cards of the same type. Here we introduce an iterative control structure which recognizes such an uninterrupted succession.

We have defined the regular expression `"{α}"` as the 0 or more repeated α. Here extending this definition, we define the expression `"^{α}"` as the 0 or more repeated α of which length is as long as possible. Therefore it is considered that the push-down automaton defined by the following context-free rules can recognize only the uninterrupted successions of `T1` and `T3` cards.

```
InputB->"",
InputB->T1:^{T1}:InputB,
InputB->T3:^{T3}:InputB.
```

If the input sequence is given as `"T1:T1:T1:T3:T3"`, it is always analyzed as the following derivation tree.

```
InputB
   |
   T1:T1:T1:InputB
                |
                T3:T3:InputB
                         |
                         ""
```

It can never be analyzed as follows,

```
InputB
  |
  T1:T1:InputB
          |
          T1:InputB
                |
                T3:T3:InputB
                        |
                        ""
```

because "^{T1}" means the sequence of T1 whose length must be as long as possible. If the control structure "^{...}" appears twice in an expression such as "^{T1}:^{T1}", the left one has a priority to the right one.

By using this control structure, the subprogram to count B can be written as follows,

```
{ InputB->"",               B->0 }
{ InputB->T1:^{T1}:InputB, B->1+B }
{ InputB->T3:^{T3}:InputB, B->1+B }
```

or

```
{ InputB->"",          B->0 }
{ InputB->T1s:InputB, B->1+B }
{ InputB->T3s:InputB, B->1+B }
{ T1s->T1:^{T1} }
{ T3s->T3:^{T3} }.
```

This subprogram does not use the notion of state transitions. In the same way, the subprogram to count D is rewritten as follows,

```
{ InputD->"",                D->0 }
{ InputD->T1:^{T1}:InputD,   D->D }
{ InputD->T3:^{T3}:InputD,   D->1+D }.
```

It is not so difficult to implement this control structure in the Gramp interpreter.

3.4.Loop fusion

We find that both structures of the above subprogram to count B and D are quite similar except "B->1+B" and "D->D" in each second rule-set, respectively. Further more, the subprogram to count C can be rewritten as the following subprogram similar to the above two,

```
{ InputC->"",                C->0 }
{ InputC->T1:^{T1}:InputC,   C->1{+1}+C }
{ InputC->T3:^{T3}:InputC,   C->C }
```

Since the symbols InputB, InputC and InputD are bound by InputB==InputC==InputD, these three subprograms can be merged in one subprogram which expresses the relation between four nonterminal symbols InputBCD, B, C and D, and then InputB==InputC==InputD can be replaced by one nonterminal symbol InputBCD. Finally the program is given as follows,

```
main problem4_2
```

```
...
{ CardSeq->InputA:T1:InputBCD:T2, ... }
...
{ InputBCD->"",                  B->0,    C->0,         D->0 }
{ InputBCD->T1:^{T1}:InputBCD, B->1+B, C->1{+1}+C, D->D }
{ InputBCD->T3:^{T3}:InputBCD, B->1+B, C->C,         D->1+D }
...
end
```

The program transformation technique from the naive program `problem4` into the above program `problem4_2` is similar to the so-called *loop fusion* well known in the field of functional programming [8]. Some other program transformation techniques in CCFG are presented in [5].

At last the final versions of the program to solve the problem 4 is illustrated in figure 2.

4.The Magic Mailing Company

Next we solve the problem 7 in [1].

Problem 7 The Magic Mailing Company has just sent out nearly 10000 letters containing an unrepeatable and irresistible offer, each letter accompanied by a returnable acceptance card. The letters and cards are individually numbered from 1 to 9999. Not all of the letters were actually sent, because some were spoilt in the addressing process. Not all of the people who received letters returned the reply card. The marketing manager, who is a suspicious fellow, thinks that his staff may have stolen some of the reply cards for the letters which were not sent, and returned the cards so that they could benefit from the offer.

The letter sent have been recorded on a file; there is one record per letter sent, containing the

```
main problem4
{ ?Input("inputFile")==CardSeq, Output==?LinePrinter }

{ CardSeq->InputA:T1:InputBCD:T2,
  Output->T1:T2:"A=":!charStr(A):\cr\:
               "B=":!charStr(B):\cr\:
               "C=":!charStr(C):\cr\:
               "D=":!charStr(D):\cr\ }

{ InputA->"",             A->0 }
{ InputA->T2orT3:InputA, A->1+A }

{ InputBCD->"",                  B->0,    C->0,         D->0 }
{ InputBCD->T1:^{T1}:InputBCD, B->1+B, C->1{+1}+C,  D->D }
{ InputBCD->T3:^{T3}:InputBCD, B->1+B, C->C,         D->1+D }

{ T2orT3->T2|T3 }

{ T1->"1":CharString:\cr\ }
{ T2->"2":CharString:\cr\ }
{ T3->"3":CharString:\cr\ }
end
```

Figure 2. The final version of the program to solve the problem 4

letter-number and other information which does not concern us. The reply cards are machine readable, and have been copied to tape; each reply card returned gives rise to one record containing the letter-number and some other information which again does not concern us. Both the letter file and the reply file have been sorted into letter-number order.

A program is needed to report on the current state of the offer. Each line of the report corresponds to a letter-number; there are four types of line, each containing the letter-number, a code and a message. The four types of line are:

NNNNN 1 LETTER SENT AND REPLY RECEIVED
NNNNN 2 LETTER SENT, NO REPLY RECEIVED
NNNNN 3 NO LETTER SENT, REPLY RECEIVED
NNNNN 4 NO LETTER SENT, NO REPLY RECEIVED

Two programs to solve this problem in different manners are given in figure 3 and figure 4. In the programs four new notions, record data type, `"let"`, `"where"` and `"otherwise"`, are used. They are explained in the following subsections.

The program in figure 3 is effective for the bottom-up interpreter. Contrarily, the program effective for the top-down interpreter is given in figure 4. The comparisons of the execution by the bottom-up interpreter with the top-down interpreter are discussed in the subsection 4.4.

4.1.Record data type

Gramp can define a term of the form `record(...)` as a *record data type*, which is declared by the singleton rule-set as follows,

```
{ ARecordType->record(field1->TypeOfField1,field2->TypeOfField2,...) }
```

where `ARecordType` is the name of the record data type and its field identifier names are `field1`, `field2`, ... whose types are `TypeOfField1`, `TypeOfField2`, ... The types of the fields are defined by rule-sets as follows,

```
{ TypeOfField1->... }
{ TypeOfField2->... }
...
```

For example, the nonterminal symbols `Letter` and `Reply` in figure 3 express the record data types of a letter and a reply card, respectively, and their fields are `number`, `name` and `address`.

Each field is referenced by its field identifier followed by the record type name and a dot `"."` just like `"ARecordType.field1"`. For example, there appears the rule-set of the following form in figure 3.

```
{ ..., Letters->Letters:Letter, ..., Line+1==Letter.number }
```

This rule-set means that `Letters` consists of the last `Letter` following `Letters` and that the value of the `number` field in the `Letter` is equal to the value of the `Line` plus one. This is considered to be the abbreviation of the following rule-sets

```
{ ..., Letters->Letters:record(Number,Name,Address,...), ..., Line+1==Number }
{ Number->Integer }
{ Name->CharString }
```

```
main problem7
{ let     Line==9999,
          Letters==?InputFile("letters"),
          Replies==?InputFile("replies"),
          Report==?LinePrinter }                                          ...(4.1)

{ Letter->record(number->Integer,name->CharString,address->CharString) }
                                                                          ...(4.2)
{ Reply ->record(number->Integer,name->CharString,address->CharString) }
                                                                          ...(4.3)

{ Line->0,  Letters->"",  Replies->"",
  Report->"" }                                                            ...(4.4)
{ Line->Line+1,  Letters->Letters:Letter,  Replies->Replies:Reply,
  Report->Report:
          !charStr(Line+1):" 1 LETTER SENT AND REPLY RECEIVED":\cr\,
  where   Line+1==Letter.number==Reply.number }                           ...(4.5)
otherwise
{ Line->Line+1,  Letters->Letters:Letter,  Replies->Replies,
  Report->Report:
         !charStr(Line+1):" 2 LETTER SENT,NO REPLY RECEIVED":\cr\,
  where   Line+1==Letter.number }                                         ...(4.6)
otherwise
{ Line->Line+1,  Letters->Letters,  Replies->Replies:Reply,
  Report->Report:
          !charStr(Line+1):" 3 NO LETTER SENT,REPLY RECEIVED":\cr\,
  where   Line+1==Reply.number }                                          ...(4.7)
otherwise
{ Line->Line+1,  Letters->Letters,  Replies->Replies,
 Report->Report:
         !charStr(Line+1):" 4 NO LETTER SENT,NO REPLY RECEIVED":\cr\ }
                                                                          ...(4.8)
end
```

Figure 3. The bottom-up program to solve the problem 7

```
{ Address->CharString }
```

By using this notations, we can use record data types in Gramp in the same way as in Pascal.

4.2. "where" and "let"

Conditions whether a rule-set can be applied or not are represented by left-hand-omitted-rules of the form "...==..." in the rule-set. For example, the following rule-set

```
{ Line->Line+1,  Letters->Letters:Letter, ...,
  Line+1==Letter.number }
```

means that `Line` and `Letters` are rewritten as `Line+1` and `Letters:Letter`, respectively, if it holds that the value of `Line+1` is equal to that of `Letter.number`. In order to emphasize the meaning of the left-hand-omitted-rule, the word "`where`" may be inserted before the rule as follows,

```
{ Line->Line+1,  Letters->Letters:Letter, ...,
  where   Line+1==Letter.number}
```

Although "`where`" has no meaning for the program execution, the latter rule-set is more readable than the former.

```
main problem7_2
{ let     Line==1,
          Letters==?InputFile("letters"),
          Replies==?InputFile("replies"),
          Report==?LinePrinter }                                          ...(4.9)

{ Letter->record(number->Integer,name->CharString,address->CharString) }
                                                                          ...(4.10)
{ Reply ->record(number->Integer,name->CharString,address->CharString) }
                                                                          ...(4.11)

{ Line->10000,  Letters->"",  Replies->"",
  Report->"" }                                                            ...(4.8)
{ Line->Line-1,  Letters->Letter:Letters,  Replies->Reply:Replies,
  Report->!charStr(Line-1):" 1 LETTER SENT AND REPLY RECEIVED":\cr\:
          Report,
  where   Line-1==Letter.number==Reply.number }                           ...(4.9)
otherwise
{ Line->Line-1,  Letters->Letter:Letters,  Replies->Replies,
  Report->!charStr(Line-1):" 2 LETTER SENT,NO REPLY RECEIVED":\cr\:
          Report,
  where   Line-1==Letter.number }                                         ...(4.10)
otherwise
{ Line->Line-1,  Letters->Letters,  Replies->Reply:Replies,
  Report->!charStr(Line-1):" 3 NO LETTER SENT,REPLY RECEIVED":\cr\:
          Report,
  where   Line-1==Reply.number }                                          ...(4.11)
otherwise
{ Line->Line-1, Letters->Letters, Replies->Replies,
  Report->!charStr(Line-1):" 4 NO LETTER SENT,NO REPLY RECEIVED":\cr\:
          Report }                                                        ...(4.12)
end
```

Figure 4. The top-down program to solve the problem 7

In the same sense as "`where`", the word "`let`" may be inserted before the left-hand-omitted-rule if it can be regarded as an initial assignment of a value to a nonterminal symbol at the beginning of the program execution. For example, in figure 3 we can see that the following three rule-sets express the 9999 times iteration.

```
{ Line==9999, ... }
{ Line->0, ... }
{ Line->1+Line, ... }
```

The first rule-set means the initial assignment of `9999` to `Line`, and the second means the completion condition of the iteration for the value of `Line`, and the third means the count-down procedure for the value of `Line`. In this case, the first rule-set may be rewritten as follows,

```
{ let Line==9999, ... }
```

The intuitive meaning of the left-hand-omitted-rule becomes clearer.

4.3."otherwise"

Often we want to write the control structure: "`if e1==e2 then` *<then-part>* `else` *<else-part>*" in a program. As for Gramp, it is expressed as follows,

```
{ <then-part>, where e1==e2 }
```

```
{ <else-part>, where e1!=e2 }.
```

Here != is an inequality symbol, the details of which are not discussed in this paper. However it is quite redundant to write two similar equations "e1==e2" and "e1!=e2" in a program and inefficient to evaluate the similar equations twice.

A new control mechanism is introduced by using a word "otherwise" as follows,

```
{ <then-part>, where e1==e2 }
otherwise
{ <else-part> }
```

At the time when both rule-sets are applicable, the Gramp interpreter first tries to apply the rule-set above the "otherwise". If both sides of e1==e2 are further replaced by a same *ground* term, the interpreter never retry to apply the other one under the "otherwise" even if back-tracks arise after. If both sides of e1==e2 can never be replaced by any same ground term, the interpreter does apply the other one next at the time a back-track arises. The similar mechanism can be seen in the Edinburgh Prolog [7] as a cut "!". A rule-set written above an "otherwise" has at least one left-hand-omitted-rule with a "where".

In figure 3, there are three "otherwise"s, which express "if ... then ... elseif ... then ... elseif ... then ... else ...".

4.4.Bottom-up solving and top-down solving

At last, we see that the Gramp program in figure 3 solves the problem 7. This program use the newly introduced notations, record data types, "where", "let" and "otherwise". The rule-set (4.1) is the main rule-set which connects nonterminal symbols with the system's input/output devices. The rule-set (4.2) and (4.3) define the record data types of Letter and Reply. The rule-set (4.4) defines the termination condition of iteration that it terminates if both sequence Letters and Replies become empty and the line number (the value of Line) is zero. The rule-sets from (4.5) to (4.8) define the four kind of processing context. The rule-set (4.5) expresses the case 1 that Letters and Replies contain Letter and Reply, respectively, whose numbers are equal to the line number. The rule-set (4.6) expresses the case 2 that Letters contains Letter whose number is equal to the line number but that Replies does not contain such Reply. The rule-set (4.7) expresses the case 3 that Replies contains Reply whose number is equal to the line number but that Letters does not contain such Letter. The rule-set (4.8) expresses the case 4 that neither Letters nor Replies contains an element whose number is equal to the line number. In each case, the appropriate message is added to the Report.

In figure 3, we represent the iteration of reading a letter record as follows,

```
{ let Line==9999, Letters==?InputFile("letters"), ... }
{ Line->0,        Letters->"", ... }
{ Line->Line+1,   Letters->Letters:Letter, ... }
...
```

On the other hand, we can also represent it as follow,

```
{ let Line==1,    Letters==?InputFile("letters"), ... }
{ Line->10000,    Letters->"", ... }
```

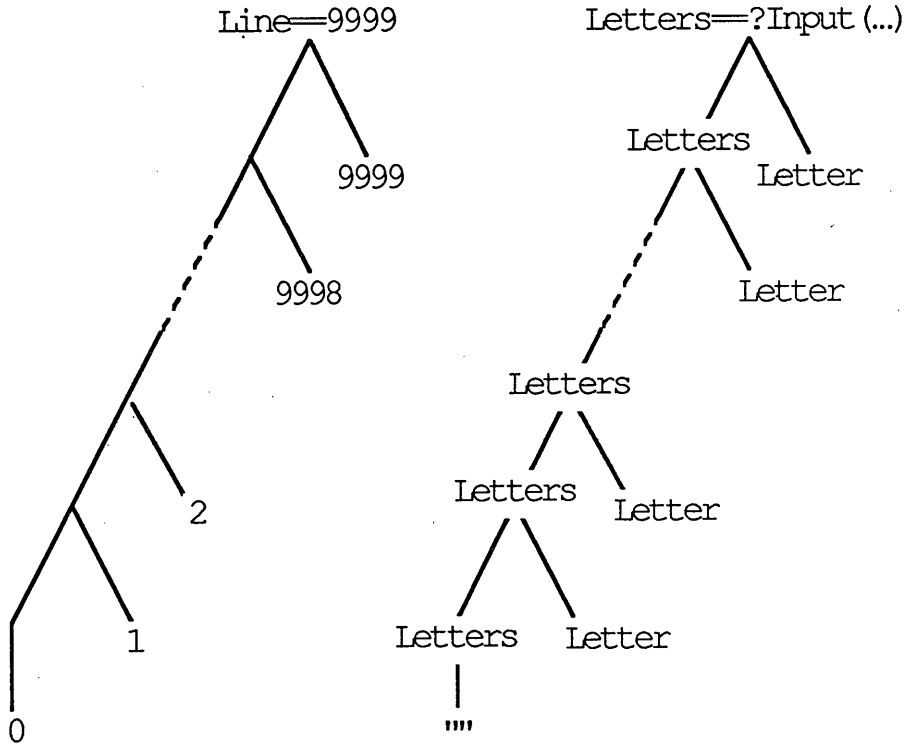

Figure 3

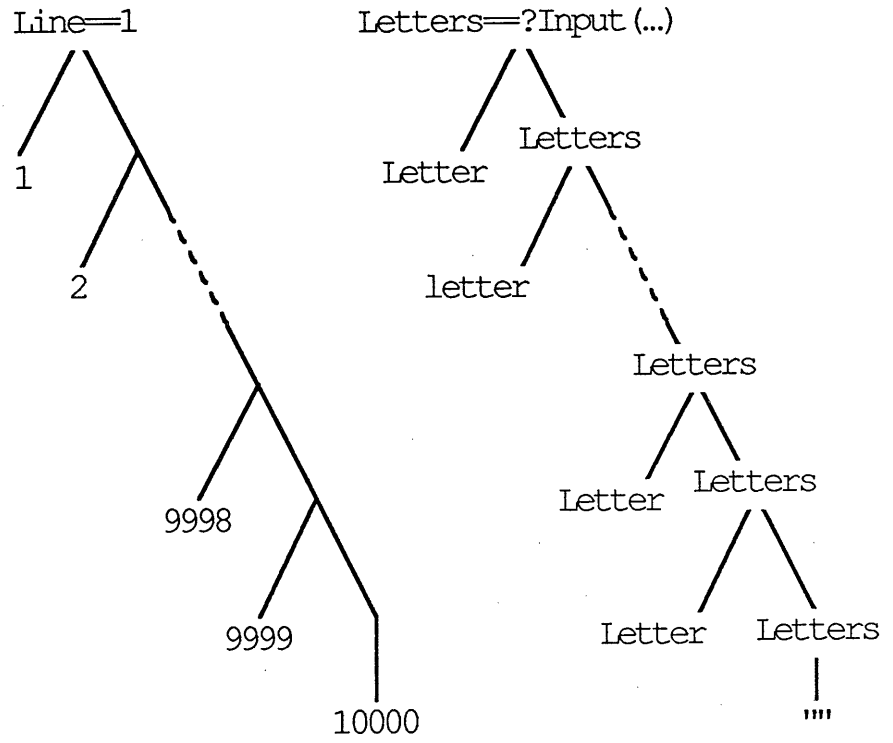

Figure 4

Figure 5. Derivation trees for the Gramp programs in figure 3 and 4

```
{ Line->Line-1,   Letters->Letter:Letters, ... }
...
```

The former definition of `Letters` is left-recursive and is not suitable for being executed by a *top-down* Gramp interpreter which reads the input sequence from left to right because many back tracks arise (recall the actions of top-down parser on context-free grammars [6]). It is suitable for a *bottom-up* interpreter (recall the actions of bottom-up parser [6]). Contrarily the latter definition is not left-recursive and is suitable for a top-down interpreter. By looking one record ahead, it is decidable which production rule should be applied, `Letters->""` or `Letters->Letter:Letters`. If the bottom-up Gramp interpreter executes the latter programs, it consumes much stack memories (recall the actions of bottom-up parser).

Thus the former program is said to be a bottom-up program and the latter is said to be a top-down one. The structures of these two kind of programs are illustrated in figure 5 by using derivation trees.

5.Telegram analysis

Problem 13 An input file on paper tape contains the texts of a number of telegrams. The tape is accessed by a "read block" instruction, which reads into main storage a variable-length character string delimited by a terminal EOB character: the sizeof a block cannot exceed 100 characters, excluding the EOB. Each block contains a number of words, separated by space character; there may be one or more spaces between adjacent words, and at the beginning and end of a block there may (but need not) be one or more additional spaces. Each telegram consists of a number of words followed by the special word "ZZZZ"; the file is terminated by a special end-file block, whose first character is EOF. In addition, there is always a null telegram at the end of the file, in the block preceding the special end-file block: this null telegram consists only of the word "ZZZZ". Except for the fact that the null telegram always appears at the end of the file, there is no particular relationship between blocks and telegrams: a telegram may begin and end anywhere within a block, and may span several blocks; several telegrams may share a block.

The processing required is an analysis of the telegrams. A report is to be produced showing for each telegram the number of words it contains and the number of those words which are oversize (more than 12 characters). For purposes of the report, "ZZZZ" does not count as a word, nor does the null telegram count as a telegram. The format of the report is:

```
TELEGRAM ANALYSIS
TELEGRAM1     15 WORDS OF WHICH 2 OVERSIZE
TELEGRAM2     106 WORDS OF WHICH 13 OVERSIZE
TELEGRAM3     42 WORDS OF WHICH 0 OVERSIZE
...
END ANALYSIS
```

The Gramp program to solve this problem is given in figure 6

```
main problem13
{ ?PaperTape==BlockFile,
              NoEOBBlockFile==TelegramFile,
                              Report==?LinePrinter }                    ...(5.1)

{      BlockFile->     {Block}:     LastBlock:     EndFileBlock,
  NoEOBBlockFile->{NoEOBBlock}:NoEOBLastBlock:NoEOBEndFileBlock }
                                                                        ...(5.2)
{      Block->CharString:\EndOfBlock\,
  NoEOBBlock->CharString:\space\ }                                      ...(5.3)

{      LastBlock->CharString:"ZZZZ":\EndOfBlock\,
  NoEOBLastBlock->CharString:\space\ }                                  ...(5.4)

{      EndFileBlock->\EndOfFile\,
  NoEOBEndFileBlock->"" }                                               ...(5.5)


{ TelegramFile->Telegrams:Spaces,
  Report->"TELEGRAM ANALYSIS":\cr\:ReportBody:"END ANALYSIS",
  with          TelegramsCount }                                        ...(5.6)

{ Telegrams->"", ReportBody->"", TelegramsCount->0 }                    ...(5.7)
{ Telegrams->Telegrams:Spaces:Telegram,
  ReportBody->ReportBody:
              "TELEGRAM":!charStr(TelegramsCount+1):\tab\:ReportLine,
  TelegramsCount->TelegramsCount+1 }                                    ...(5.8)

{  Telegram->Words:Spaces:"ZZZZ",
   ReportLine->!charStr(WordsCount):" WORDS OF WHICH ":
               !charStr(OverSizeWordsCount):" OVERSIZE":\cr\ }          ...(5.9)

{ Words->"", WordsCount->0, OverSizeWordsCount->0 }                     ...(5.10)
{ Words->Word:Spaces:Words, WordsCount->WordsCount+1,
  OverSizeWordsCount->OverSizeWordsCount,
  where   WordSize < 13 }                                  ...(5.11)
otherwise
{ Words->Word:Spaces:Words, WordsCount->WordsCount+1,
  OverSizeWordsCount->OverSizeWordsCount+1 }                            ...(5.12)

{ Word->{AlphaNumeric}, WordSize->0{+1} }                               ...(5.13)
{ Spaces->\space\:{\space\} }                                           ...(5.14)
end
```

Figure 6. The program to solve the problem 13

5.1.Intermediate file

It is known that this problem should be solved by using an intermediate file because of the *structure clash* between the input file of blocks and the format of the report. In Gramp such an intermediate file is implemented by a left-hand-omitted-rule as in the main rule-set (5.1) in figure 6. Here `?PaperTape` is the special nonterminal symbol which expresses a papar-tape device in our system. The nonterminal symbol `BlockFile` means the sequence of blocks which are separated by `\EndOfBlock\` marks. Eliminating all `\EodOfBlock\` marks, it is translated into the sequence `NoEOBBlockFile` of telegrams which are separated by the special words `"ZZZZ"`. This is connected with `TelegramFile` by using ==. The sequence `Report` is obtained by analyzing `TelegramFile`, and it is printed out to the line printer device.

Hence the Gramp program to solve this problem consists of three parts, the main rule-set

(5.1), the subprogram (the rule-sets from (5.2) to (5.5)) to translate `BlockFile` to `NoEOBBlockFile`, and the subprogram (the rule-sets from (5.6) to the last) to obtain `Report` from `TelegramFile`.

5.2."with"

In the previous section, we have introduced the words `"where"` and `"let"` in order to make Gramp programs more readable. One more word `"with"` is introduced here, which is followed by one or more nonterminal symbols as follows,

```
{ ...->...Telegrams..., ...->..ReportBody..., with TelegramsCount }        ...(5.6)
```

in the same way as in the rule-set (5.6) in the figure 6. This rule-set has the same meaning as the following rule-set,

```
{ ...->...Telegrams..., ...->..ReportBody..., where TelegramsCount==Any }    ...(5.6)´
```

or

```
{ let TelegramsCount==Any, ...->...Telegrams..., ...->..ReportBody... }.    ...(5.6)´´
```

Since `Any` means an arbitrary data object, the value of the `TelegramsCount` is not bound. However it is required that there appears `TelegramsCount` in the right-hand sides in the rule-set, because the three nonterminal symbols `Telegrams`, `ReportBody` and `TelegramsCount` must be *simultaneously* rewritten by the rule-set (5.7) or (5.8). In this case, we should use the word `"with"`.

Of course we can write the rule-set as

```
{    ...->...Telegrams..., ...->..ReportBody..., TelegramsCount},
```

(5.6)´ or (5.6)´´. However the meaning of `TelegramsCount` is not clearly readable from the text.

6.Comparison of Gramp with Prolog

Arbitrary Prolog program can automatically be translated into the equivalent Gramp program and vice versa [3]. This transformation shows us that the mathematical structures of both programs are in the dual relation. Therefore the results discussed in one paradigm are also available in the other paradigm. However the mathematical relationship does not always imply the similar programming styles and techniques. In this section we overview what kind of problems are suitable for solving in Gramp and what for solving in Prolog.

Our first suggestion is that the problem whose input/output data structures are hierarchical and easily written in context-free grammars are more suitable for Gramp than Prolog. The problem 4, 7 and 13 described above are such problems.

For example, the Gramp program in figure 2 is translated into the Prolog program in figure 7. Here we assume that this Prolog, just like Prolog III [10], can treat strings and rational numbers in the same way as terms, the predicate symbol `charString` is system-defined and an atom `charString(X)` is true if `X` is a character string, and that `!charStr(...)` is a function-call same as in the figure 2. Several variable names in the figure 7 are the same as the nonterminal symbol names appearing in the figure 2. Since a regular expression can not be used in Prolog, it is expressed by

```
?-input("inputFile",X),problem4(X,Y),linePrinter(Y).

problem4(InputA:T1:InputBCD:T2,
         T1:T2:
         "A=":!charStr(A):\cr\:
         "B=":!charStr(B):\cr\:
         "C=":!charStr(C):\cr\:
         "D=":!charStr(D):\cr\):-t1(T1),t2(T2),processA(InputA,A),
                                 processBCD(InputBCD,B,C,D).

processA("",0).
processA(X:InputA,1+A):-t2ort3(X),processA(InputA,A)

processBCD("",0,0,0).
processBCD(X,1,Y,1):-processT1s(X,Y).
processBCD(X:Y:InputBCD,1+B,Z+C,1+D):-processT1s(X,Z),t3(Y),
                                      processBCD(Y:InputBCD,B,C,D).
processBCD(T3s,1,0,1):-t3s(T3s).
processBCD(X:Y:InputBCD,1+B,C,1+D):-t3s(X),t3(Y),
                                    processBCD(Y:InputBCD,B,C,D).

processT1s(X,1):-t1(X).
processT1s(X:Y,1+Z):-t1(X),processT1s(Y,Z).
t2ort3(X):-t2(X);t3(X).
t3s(X):-t3(X).
t3s(X:Y):-t3(X),t3s(Y).

t1("1":X:\cr\):-charStr(X).
t2("2":X:\cr\):-charStr(X).
t3("3":X:\cr\):-charStr(X).
```

Figure 7. The Prolog program to solve the problem 4

several clauses.

In the Gramp program, it is easy to read each input/output data structure by extracting the context-free rules related to the data structure.

```
InputA->"",
InputA->T2orT3:InputA.
```

From these two rules we can easily understand that `InputA` is a sequence of `T2orT3`. On the other hand, in the Prolog program the structure of the first argument of the predicate `proceesA(...)`, which corresponds to the nonterminal symbol `InputA`, is obtained by abstracting second argument parts of the atoms as follows,

```
processA("",...).
processA(X:InputA,...):-t2ort3(X),processA(InputA,...)
```

We see that the data structure is directly readable from the former context-free rules while indirectly from the latter definite clauses.

Our second suggestion is that the problems in which there are a number of complicated relations rather than their hierarchical data structures are suitable for Prolog than Gramp. For example, there is a well known problem about ancestors-descendant relation written in Prolog as follows,

```
ancestor(X,Y):-father(X,Y).
```

```
ancestor(X,Z):-father(X,Y),ancestor(Y,Z).
father(a,b).
father(b,c).
...
```

This program is translated into the following Gramp program,

```
{ Ancestor->Father, Descendant->Child }
{ Ancestor->Father, Descendant->Descendant,
  where   Child==Ancestor }
{ Father->a,    Child->b }
{ Father->b,    Child->c }
...
```

This Gramp program is not so readable as the Prolog program, because the relations can not be expressed well. In this case, the input/output data structures are simple and the context-free rules which express the data structure of `Ancestor` are as follows

```
Ancestor->Father
Ancestor->Father,
Father->a,
Father->b,
...
```

which show almost nothing.

7.Concluding remarks

In this paper we have introduced a programming language: Gramp based on CCFG and shown that Gramp is effective to express the programs whose input/output data objects have some kind of structures, especially hierarchical ones in the sense of the Jackson program development method. Although Gramp and Prolog stand on a common mathematical basis, their syntaxes are quite different from each other. Hence a programmer should choose the effective one according to a problem to solve.

We are now developing the Gramp interpreter which can execute the program described in this paper.

Reference

[1] Jackson, M. A. : *Principles of Program Design*, Academic Press (1975).

[2] Yamashita, Y. and Nakata, I. : Programming in Coupled Context-Free Grammars, submitted for publication.

[3] Yamashita, Y. and Nakata, I. : On the Relation between CCFG Programs and Logic Programs, submitted for publication.

[4] Yamashita, Y. and Nakata, I. : The Unified Theory of CCFG Programs with Logic and

Functional Programs, Institute of Electronics, Information and Communication Engineers, Workshop Report on Computation, COMP 87-40 (1987), (*In Japanese*)

[5] Nakata, I. and Yamashita, Y. : Program Transformations in Coupled COntext-Free Grammar, *Computer Software*, to appear (1988). (*In Japanese*)

[6] Aho, A. V. and Ullman, J. D. :*The Theory of Parsing, Translation, and Compiling*, Volume I: Parsing, Prentice-Hall (1972).

[7] Clocksin, W. F. and Mellish, C. S. : *Programming in Prolog*, Springer-Verlag (1981).

[8] Burstall,R.M. and Darlington,J. : A Transformation System for Developing Recursive Programs, J.ACM 24(1977), pp.44-67.

[9] Lalonde, W. R. : Regular Right Part Grammars and their Parsers, C. ACM, Vol.20, No.10 (1977), pp.731-741.

[10] Colmerauer, A. : Opening the Prolog III Universe, *BYTE*, August (1987), pp.177-195.

INSTITUTE OF INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA
SAKURA-MURA, NIIHARI-GUN, IBARAKI 305 JAPAN

REPORT DOCUMENTATION PAGE

REPORT NUMBER
ISE-TR-88-73

TITLE

Programming in Gramp: a programming language based on CCFG

AUTHOR(S)

Yoshiyuki YAMASHITA,
Ikuo NAKATA,

REPORT DATE
Jun. 1, 1988

NUMBER OF PAGES
22

MAIN CATEGORY
Programming Languages

CR CATEGORIES
D.1, D.3

KEY WORDS

Programming Language, Coupled Context-Free Grammar,
Data Structure Directed Programming, The Jackson Method

ABSTRACT

Gramp is a declarative programming language based on Coupled Context-Free Grammar (CCFG). It is considered that Gramp is effective to express the programs whose input/output data objects have some kind of structures, especially hierarchical ones. In this sense CCFG formalizes and Gramp embodies the Jackson program design. Here some problems described in [1] are expressed in Gramp programs. It is shown that the programs are more readable than in other languages. Several ideas of syntax-sugaring and control in Gramp are also discussed.

[1] Jackson, M. A. : *Principles of Program Design*, Academic Press (1975).

SUPPLEMENTARY NOTES