A CONCISE EXTENSIBLE METALANGUAGE FOR TRANSLATOR IMPLEMENTATION*

DOUGLAS L. MICHELS
Scotts Valley, California

A very concise metalanguage is presented. This language is capable of describing context free languages, including itself. Several mutually recursive functions define an interpreter for this language. The metalanguage and interpreter are extended to allow the inclusion of emitters. This makes possible the description of translations. A metatranslater is shown which is capable of self-translation. The addition of labeled productions makes possible a metatranslation language in the style of BNF.

Key words and phrases: metalanguage, metatranslator, metacompiler, self-describing grammar

1. INTRODUCTION

McKeeman (4) has suggested a refinement approach to the construction of translator writing systems. This approach is based on partitioning the system into several languages, one for each major component of the resultant translators. Instead of constructing the translator writing system in its totality, it is to be "evolved", each generation a product of the tools created by the previous generation. The design objective for each generation is the creation of the most useful tools with which to "evolve" the next generation.

McKeeman has named the basis step in this evolution the SEED. The seeds of a translator writing system are the tools necessary to create the minimal translators required for the description of more sophisticated languages. An ideal seed would have the capability to build several very simple but significantly different translators. The seed and the languages constructed with it serve only as development steps and therefore execution efficiency is far less important than conceptual clarity and extensibility.

Schorre (7) proposed a meta translation system which added output rules to BNF (6) style syntax equations. This class of translators is of sufficient capability to be used as a seed; but appears to be more complex then necessary. This complexity results from the structure of the generated machine language, which requires the compiler to generate addresses for branching. The predefined scanner adds rigidity and additional complexity.

We propose that a meta translation system based on a prefix operator machine language is conceptually simple and easy to implement. In addition scanning is viewed as yet another translation and is not defined in the system.

*This research was performed as an Information Science student at the University of California, Santa Cruz, California and was supported in part by ONR Contract No. N00014-76-C-0682.

2. GO: A SELF-RECOGNIZING RECOGNIZER

A very concise meta-language (GO) can be defined using only four prefix operators. This language can recognize context free languages. To demonstrate this the syntax of the GO is expressed using GO.

GO contains two binary operators, concatenation ('&') and alternation ('|'). The '&' is true if both of the rules which follow it are true. If either of these operators are false the input is backed up to the point preceding that operator's evaluation. A unary literal operator ('"') is provided to test the next character of input. The '"' is true if the character which follows it is identical to the next character of input. The input character is consumed from the input string. The recursion operator ('.') is used to re-invoke the entire grammar. It is true only if the first operator in the grammar is true. The full specification of the machine which executes this grammar is given in section 3.

Figure 2.0 shows the syntax of GO in a BNF-like language. Non-terminals are defined by single symbols on the left of '=', terminals are quoted by single quotes '''. Alternate productions are separated by '|' and concatenation is denoted by juxtaposition of rules. Parentheses are used to alter the normal precedence. The first non-terminal is the start symbol. This notation will be used throughout the paper and a full grammar for it given in a later section.

```
S = '.'
  | ( '&' | '|' ) S S
  | '"' ( '&' | '|' | '"' | '.' )
  ;
```

Figure 2.0: GO in BNF-like notation

```
l ".
  l & l "&
        "l
      & .
        .
    & ""
      l "&
        l "l
          l ""
            ".
```

Figure 2.1: GO in GO

3. MO: AN INTERPRETER FOR GO

A machine which will directly execute GO can be defined by several simple, mutually recursive functions. In order to define these functions several primitive string operators are required. The functionality and function of each is described:

first: STRING -> STRING
first (S) results in the left-most character of S.

rest: STRING -> STRING
rest (S) results in S with first (S) deleted.

equal: STRING X STRING -> BOOLEAN
equal (S1,S2) is TRUE if and only if S1 and S2 are identical.

Machine is a BOOLEAN function. Machine (Grammar, Input) is TRUE if and only if Input is in the language defined by Grammar. The following functions define Machine:

```
Machine (G,I) =
  IF Test (G,G,I) AND
     equal(Remaining (G,G,I), NULL)
     THEN
        TRUE
     ELSE
        FALSE
END Machine

Test (G,R,I) =
 CASE first (R) OF
 '.' : Test (G,G,I)
 '&' : IF Test (G,rest(G),I)
          THEN
             Test (G,Skip(rest (R)),
             Remaining(G,rest(R),I))
          ELSE
             FALSE
 'l' : IF Test (G,rest(R),I)
          THEN
             TRUE
          ELSE
             Test (G,Skip(rest(R)),I)
 '"' : equal (first(rest(R)),first(I))
 END CASE
END Test

Remaining (G,R,I) =
 CASE first (R) OF
 '.' : Remaining (G,G,I)
 '&' : Remaining (G,Skip(rest(R)),
       Remaining(G,rest(R),I))
 'l' : IF Test(G,rest(R),I)
          THEN
             Remaining(G,rest(R),I)
          ELSE
             Remaining(G,Skip(rest(R)),I)
 '"' : rest(I)
 END CASE
END Remaining

Skip (R) =
  CASE first (R) OF
  '.' : rest (R)
  '&' : Skip(Skip(rest(R)))
  'l' : Skip(Skip(rest(R)))
  '"' : rest(rest(R))
  END CASE
 END Skip
```

Figure 3.0: Machine MO, executes GO

4. G1: A SELF-TRANSLATING TRANSLATOR

In order to perform translations it is necessary to augment GO with emitters. This is done with the emit operator, ('>'). This operator functions much like the literal operator, except that it outputs the character which follows it. The emit operator is always true.

```
S = '.'
  l ( '&' l 'l' ) S S
  l ( '"' l '>' ) ( '.' l '&' l 'l' l '>' )
  ;
```

Figure 4.0: G1 in BNF-like notation

```
l & ".
    >.
  l & l & "&
          >&
        & "l
          >l
      & .
        .
    & l & ""
          >"
        & ">
          >>
      l & ".
          >.
        l & "&
            >&
          l & "l
              >l
            l & ""
                >"
              & ">
                >>
```

Figure 4.1: G1 to G1 in G1

5. M1: A TRANSLATION MACHINE

To execute G1 we must not only determine if the input string is in the language, but what output should be generated if it is. A new function Emit (G, R, I) is needed. Emit (Grammar, Grammar, Input) is returned if Machine (Grammar, Input) is true. Remaining, Test, and Skip need a case added for ' '. A new string function is needed to assemble the output:

```
concat:  STRING X STRING ->STRING
         concat (S1,S2) appends S2 to S1;
         S = concat (first (S), rest (S))
```

To save space only the new function Emit is shown in figure 5.0. The other functions are modified in a way analogous to section 7.

```
Emit (G,R,I) =
 CASE first(R) OF
 '&' : concat(Emit(G,rest(R),I),
       Emit(G,Skip(rest(R)),
       Remaining(G,rest(R),I)))
 '|' : If Test (G,rest(R),I)
          THEN
             Emit (G,rest(R),I)
          ELSE
             Emit (G,Skip(rest(R)),I)
 '>' : first(rest(R))
 '"' : NULL
 END CASE
END EMIT
```

Figure 5.0: M1 Emit function

6. G2: BELLS AND WHISTLES

In order to specify languages in a production system format we will need the ability to label productions. A new operator (':') is introduced. This operator is a unary prefix operator and is true if the rule labeled by the character which follows it is true. It essentially is a subroutine call.

This capability makes possible a BNF-like notation to G2 translator expressed in itself. Output strings are delimited by brackets ('{','}'). One such translator is shown in figure 6.1.

```
G = L R G
  | L R
  ;

R = ( ':' | '"' | '>' ) L
  | ( '&' | '|' ) R R
  ;

L = 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
  | 'G' | 'H' | 'I' | 'J' | 'K' | 'L'
  | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
  | 'S' | 'T' | 'U' | 'V' | 'W' | 'X'
  | 'Y' | 'Z'
  | '=' | ';' | '(' | ')' | '|' | '&'
  | '>' | ':' | '"' | ' ' | '{'
  ;
```

Figure 6.0: G2 grammar expressed in BNF-like notation

```
R = L '=' A ';' R
  | L '=' A ';'
  ;

A = {:} C '|' A
  | C
  ;

C = {&} I ' ' C
  | I
  ;

I = ''' ( ''' {"'} | S ) '''
  | '{' ( O | '}' {>} {}} ) '}'
  | '(' A ')'
  | {:} L
  ;

S = {&"} ( L | '}' {}} ) S
  | {"} ( L | '}' {}} )
  ;

O = {&}  ( L | ''' {'} ) O
  | {>} ( L | ''' {'}
  ;

L = 'A' {A} | 'B' {B} | 'C' {C}
  | 'D' {D} | 'E' {E} | 'F' {F}
  | 'G' {G} | 'H' {H} | 'I' {I}
  | 'J' {J} | 'K' {K} | 'L' {L}
  | 'M' {M} | 'N' {N} | 'O' {O}
  | 'P' {P} | 'Q' {Q} | 'R' {R}
  | 'S' {S} | 'T' {T} | 'U' {U}
  | 'V' {V} | 'W' {W} | 'X' {X}
  | 'Y' {Y} | 'Z' {Z}
  | '=' {=} | ';' {;} | '(' {(}
  | ')' {)} | '|' {|} | '&' {&}
  | '>' {>} | ':' {:} | '"' {"}
  | ' ' { } | '{' {{}
  ;
```

Figure 6.1: BNF-like notation to G2 translator in BNF-like notation

7. M2: AN INTERPRETER FOR G2

The interpreter for G2 is very similar to the interpreter for G0 and G1. Besides including the additional operator ':' in each of the case statements, a simple function Find is needed to locate the start of a named rule.

```
Machine (G,I) =
 IF Test (G,rest(G),I) AND
    equal (Remaining(G,rest(G),I),NULL)
    THEN
       (TRUE,Emit (G,rest(G),I)
    ELSE
       (FALSE,NULL)
END Machine

Test (G,R,I) =
 CASE first (R) OF
 ':' : Test (G,Find(G,rest(R)),I)
 '&' : IF Test (G,rest(R),I)
          THEN
             Test (G,Skip(rest(R)),
             Remaining(G,rest(R),I))
          ELSE
             FALSE
```

```
 '1' : IF Test (G,rest(R),I)
          THEN
             TRUE
          ELSE
             Test (G,Skip(rest(R)),I)
 '>' : TRUE
 '"' : equal (first(rest(R)),first(I))
 END CASE
END Test

Remaining (G,R,I) =
 CASE first(R) OF
 ':' : Remaining (G,Find(G,rest(R)),I)
 '&' : Remaining (G,Skip(rest(R)),
       Remaining(G,rest(R),I))
 '1' : IF Test(G,rest(R),I)
          THEN
             Remaining(G,rest(R),I)
          ELSE
             Remaining(G,Skip(rest(R)),I)
 '>' : I
 '"' : rest(I)
 END CASE
END remaining

Emit (G,R,I) =
 CASE first(R) OF
 ':' : Emit (G,Find(G,rest(R)),I)
 '&" : Concat(Emit(G,rest(R),I),
       Emit(G,Skip(rest(R)),Remaining
       (G,rest(R),I)))
 '1' : IF Test (G,rest(R),I)
          THEN
             Emit (G,rest(R),I)
          ELSE
             Emit (G,Skip(rest(R)),I)
 '>' : first(rest(R))
 '"' : NULL
 END CASE
END Emit

Skip (R) =
 CASE first (R) OF
 ':' : rest(rest(R))
 '&' : Skip(Skip(rest(R)))
 '1' : Skip(Skip(rest(R)))
 '>' : rest(rest(R))
 '"' : rest(rest(R))
 END CASE
End Skip

Find (G,R) =
 IF equal (first(G),first(R))
    THEN
       rest(G)
    ELSE
       Find (Skip(rest(G)),R)
END Find
```

Figure 7.0: Machine M2 executes G2

8. LIMITATIONS

These languages have several significant limitations. The object code, while adequate for expressing many interesting languages is nearly impossible for a human reader to comprehend. All versions of the language are sensitive to the ordering of the productions. It is difficult to verify that the ordering is completely correct. The type of translations that can be produced are limited to those which do not require radical restructuring of the source code.

CONCLUSIONS

A class of grammars has been defined for which a translator can be concisely stated and simply implemented. This class of grammars is sufficiently powerful to allow the definition of more expressive languages.

Fay (2) has demonstrated that a direct implementation of the interpreter executes painfully slow. He provides an example of a similar implementation that is easily implemented and has reasonable execution characteristics.

Techniques to facilitate the creation of powerful problem oriented languages will continue to be investigated. Limiting the problem to finding the smallest useful yet implementable system has provided several important insights, as well as a possibly fertile seed for evolving more sophisticated translation systems.

AKNOWLEDGMENT

The ideas summarized in this report originated in a research group consisting of Bill McKeeman, Jim Horning, Dan Ross and Bill Fitler. Many UCSC faculty and staff provided helpful insights and suggestions. In particular the author would like to thank Frank DeRemer, Frank Frazier and Michael Fay. In addition the author is indebted to Bill McKeeman; without whom this paper would not have been started, let alone completed.

REFERENCES

{1} Chomsky, Noam, Syntactic Structures, Mouton and Co., The Hague, The Netherlands (1957).

{2} Fay, Michael, Bootstrapping a Small Translator Writing System, UCSC technical report 77-3-002, University of California at Santa Cruz, March 1977.

{3} Hopcroft, J. E., and Ullman, J. D., Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Mass. (1969).

{4} McKeeman, W. M., Private Communication, (1976).

{5} Michels, Douglas, A Concise Extensible Metalanguage for Translator Implementation, UCSC technical report 78-4-001, University of California at Santa Cruz, July 1976.

{6} Naur, P. (ed.) et al., Report on the Algorithmic Language ALGOL 60. Comm ACM, 3:5, 1960 299-314.