A BNF PARSER IN FORTH

(This article first appeared in ACM SigFORTH Newsletter vol. 2 no. 2)

Bradford J. Rodriguez
T-Recursive Technology
115 First St. #105
Collingwood, Ontario L9Y 4W3 Canada
bj@forth.org

1. Introduction

Backus-Naur Form (BNF) is a notation for the formal description of programming languages. While most commonly used to specify the syntax of "conventional" programming languages such as Pascal and C, BNF is also of value in command language interpreters and other language processing.

This paper describes a one-screen Forth extension which transforms BNF expressions to executable Forth words. This gives Forth a capability equivalent to YACC or TMG, to produce a working parser from a BNF language description.

2. BNF Expressions

BNF expressions or productions are written as follows:

```
          production  ::=  term ... term    (alternate #1)
                        |  term ... term    (alternate #2)
                        |  term ... term    (alternate #3)
```

This example indicates that the given production may be formed in one of three ways. Each alternative is the concatenation of a series of terms. A term in a production may be either another production, called a nonterminal, or a fundamental token, called a terminal.

A production may use itself recursively in its definition. For example, an unsigned integer can be defined with the productions

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> ::= <digit> | <digit> <number>
```

which says that a number is either a single digit, or a single digit followed by another number (of one or more digits).

We will use the conventions of a vertical bar | to seperate alternatives, and angle brackets to designate the name of a production. Unadorned ASCII characters are terminals (the fundamental tokens).

3. A Simple Solution through Conditional Execution

The logic of succession and alternation can be implemented in two "conditional execution" operators, && and ||. These correspond exactly to the "logical connectives" of the same names in the C language (although their use here was actually inspired by the Unix "find" command). They are defined:

```
: || IF R> DROP 1 THEN ; ( exit on true)
: && 0= IF R> DROP 0 THEN ; ( exit on false)
```

|| given a true value on the stack, exits the colon definition immediately with true on the stack. This can be used to string together alternatives: the first alternative which is satisfied (returns true) will stop evaluation of further alternatives.

&& given a false value on the stack, exits the colon definition immediately with false on the stack. This is the "concatenation" operator: the first term which fails (returns false) stops evaluation and causes the entire sequence to fail.

We assume that each "token" (terminal) is represented by a Forth word which scans the input stream and returns a success flag. Productions (nonterminals) which are built with such tokens, ||, and &&, are guaranteed to return a success flag.

So, assuming the "token" words '0' thru '9' have been defined, the previous example becomes:

```
            : <DIGIT>   '0' || '1' || '2' || '3' || '4'
                      || '5' || '6' || '7' || '8' || '9' ;
            : <NUMBER1>   <DIGIT> && <NUMBER> ;
            : <NUMBER>   <DIGIT> || <NUMBER1> ;
```

Neglecting the problem of forward referencing for the moment, this example illustrates three limitations:

a) we need an explicit operator for concatenation, unlike BNF.

b) && and || have equal precedence, which means we can't mix && and || in the same Forth word and get the equivalent BNF expression. We needed to split the production <NUMBER> into two words.

c) we have made no provision for restoring the scan pointer if a BNF production fails.

We will address these next.

4. A Better Solution

Several improvements can be made to this "rough" BNF parser, to remove its limitations and improve its "cosmetics."

a) Concatenation by juxtaposition. We can cause the action of && to be performed "invisibly" by enforcing this rule for all terms (terminals and nonterminals): Each term examines the stack on entry. if false, the word exits immediately with false on the stack. Otherwise, it parses and returns a success value.

To illustrate this: consider a series of terms

```
<ONE> <TWO> <THREE> <FOUR>
```

Let `<ONE>` execute normally and return "false." The `<TWO>` is entered, and exits immediately, doing nothing. Likewise, `<THREE>` and `<FOUR>` do nothing. Thus the remainder of the expression is skipped, without the need for a return-stack exit.

The implementation of this will be described shortly.

b) Precedence. By eliminating the && operator in this manner, we make it possible to mix concatenation and alternation in a single expression. A failed concatenation will "skip" only as far as the next operator. So, our previous example becomes:

```
: <NUMBER> <DIGIT> || <DIGIT> <NUMBER> ;
```

c) Backtracking. If a token fails to match the input stream, it does not advance the scan pointer. Likewise, if a BNF production fails, it must restore the scan pointer to the "starting point" where the production was attempted, since that is the point at which alternatives must be tried. We therefore enforce this rule for all terminals and nonterminals: Each term saves the scan pointer on entry. If the term fails, the scan pointer is restored; otherwise, the saved value is discarded.

We will later find it useful to "backtrack" an output pointer, as well.

d) Success as a variable. An examination of the stack contents during parsing reveals the surprising fact that, at any time, there is only one success flag on the stack! (This is because flags placed on the stack are immediately "consumed.") We can use a variable, `SUCCESS`, for the parser success flags, and thereby simplify the manipulations necessary to use the stack for other data. All BNF productions accept, and return, a truth value in `SUCCESS`.

5. Implementation

The final BNF parser word set is given on screen 3. Three words implement the essential logic:

`<BNF` is used at the beginning of a production. If `SUCCESS` is false, it causes an immediate exit. Otherwise, it saves the scan pointer on the return stack.

`|` separates alternatives. If `SUCCESS` is true, it causes an immediate exit and discards the saved scan pointer. Otherwise, it restores the scan position from the saved pointer.

`BNF>` is used at the end of a production. If `SUCCESS` is false, it restores the scan position from the saved pointer. In any case, it removes the saved pointer from the return stack.

`<BNF` and `BNF>` are "run-time" logic, compiled by the words `BNF:` and `;BNF`, respectively.

`BNF: name` starts the definition of the BNF production name.

`;BNF` ends a BNF definition.

Finally, there are four words which simplify the definition of token words and other terminals:

`@TOKEN` fetch the current token from the input.

`+TOKEN` advance the input scan pointer.

`=TOKEN` compare the value on top of stack to the current token, following the rules for BNF parsing words.

`nn TOKEN name` builds a "terminal" name, with the ASCII value nn.

The parser uses the fig-Forth IN as the input pointer, and the dictionary pointer DP as the output pointer. These choices were made strictly for convenience; there is no implied connection with the Forth compiler.

6. Examples and Usage

The syntax of a BNF definition in Forth resembles the "traditional" BNF syntax:

Traditional: `prod ::= term term | term term`
Forth: `BNF: prod term term | term term ;BNF`

Screen 6 is a simple pattern recognition problem, to identify text having balanced left and right parentheses. Several aspects of the parser are illustrated by this example:

a) Three tokens are defined on line 4. To avoid name conflicts, they are named with enclosing quotes. `<EOL>` matches the end-of-line character in the fig-Forth Terminal Input Buffer.

b) Line 9 shows a recursive production, `<S>`. During the definition of a production, its name is automatically unSMUDGEd.

c) Line 9 also shows a null alternative. This is often encountered in BNF. The null alternative parses no tokens, and is always satisfied.

d) Not all parsing words need be written as BNF productions. Line 6 is Forth code to parse any ASCII character, excluding parentheses and nulls. Note that `BNF:` and `;BNF` are used, not to create a production, but as an easy way to create a conditionally-executing (per `SUCCESS`) Forth word.

e) Line 11 shows how to invoke the parser: `SUCCESS` is initialized to "true," and the "topmost" BNF production is executed. on its return, `SUCCESS` is examined to determine the final result.

f) Line 11 also shows how end-of-input is indicated to the BNF parser: the sequence is defined as the desired BNF production, followed by end-of-line.

Screens 7 and 8 parse algebraic expressions with precedence. This grammar is directly from [AH077], p. 138. The use of the productions `<T'>` and `<E'>` to avoid the problem of left-recursion is described on p. 178 of that book. Note also:

a) `<DIGIT>` is defined "the hard way." It would be better to do this with a Forth word.

b) `<ELEMENT>` requires a forward reference to `<EXPRESSION>`. We must patch this reference manually.

Screens 9 through 11 show how this algebraic parser can be modified to perform code generation, coincident with the parsing process. Briefly: each alternative of a BNF production includes Forth code to compile the output which would result from that alternative. If the alternative succeeds, that output is left in the dictionary. If it fails, the dictionary pointer is "backtracked" to remove that output. Thus, as the parser works its way, top-down, through the parse tree, it is constantly producing and discarding trial output.

This example produces Forth source code for the algebraic expression.

a) The word `,"` appends a text string to the output.

b) We have chosen to output each digit of a number as it is parsed. `(DIGIT)` is a subsidiary word to parse a valid digit. `<DIGIT>` picks up the character from the input stream before it is parsed, and then appends it to the output. If it was not a digit, `SUCCESS` will be false and `;BNF` will discard the appended character.

If we needed to compile numbers in binary, `<NUMBER>` would have to do the output. `<NUMBER>` could start by placing a zero on the stack as the accumulator. `<DIGIT>` could augment this value for each digit. Then, at the end of `<NUMBER>`, the binary value on the stack could be output.

c) After every complete number, we need a space. We could factor `<NUMBER>` into two words, like `<DIGIT>`. But since `<NUMBER>` only appears once, in `<ELEMENT>`, we append the space there.

d) In `<PRIMARY>`, `MINUS` is appended after the argument is parsed. In `<FACTOR>`, `POWER` is appended after its two arguments are parsed. `<T'>` appends * or / after the two arguments, and likewise `<E'>` appends + or -.

In all of these cases, an argument may be a number or a sub-expression. If the latter, the entire code to evaluate the sub-expression is output before the postfix operator is output. (Try it. It works.)

e) `PARSE` has been modified to `TYPE` the output from the parser, and then to restore the dictionary pointer.

7. Cautions

This parser is susceptible to the Two Classic Mistakes of BNF expressions. Both of these cautions can be illustrated with the production `<NUMBER>`:

```
BNF: <NUMBER> <DIGIT> <NUMBER> | <DIGIT> ;BNF
```

a) Order your alternatives carefully. If `<NUMBER>` were written

```
BNF: <NUMBER> <DIGIT> | <DIGIT> <NUMBER> ;BNF
```

then all numbers would be parsed as one and only one digit! This is because alternative #1 -- which is a subset of alternative #2 -- is always tested first. In general, the alternative which is the subset or the "easier to-satisfy" should be tested last.

b) Avoid "left-recursion." If `<NUMBER>` were written

```
BNF: <NUMBER> <NUMBER> <DIGIT> | <DIGIT> ;BNF
```

then you will have an infinite recursive loop of `<NUMBER>` calling `<NUMBER>`! To avoid this problem, do not make the first term in any alternative a recursive reference to the production being defined. (This rule is somewhat simplified; for a more detailed discussion of this problem, refer to [AH077], pp. 177 to 179.)

8. Comparison to "traditional" work

In the jargon of compiler writers, this parser is a "top-down parser with backtracking." Another such parser, from ye olden days of Unix, was TMG. Top-down parsers are among the most flexible of parsers; this is especially so in this implementation, which allows Forth code to be intermixed with BNF expressions.

Top-down parsers are also notoriously inefficient. Predictive parsers, which look ahead in the input stream, are better. Bottom-up parsers, which move directly from state to state in the parse tree according to the input tokens, are better still. Such a parser, YACC (a table-driven LR parser), has entirely supplanted TMG in the Unix community.

Still, the minimal call-and-return overhead of Forth should alleviate the speed problem somewhat, and the simplicity and flexibility of the BNF Parser may make it the parser of choice for many applications.

9. Applications and Variations

Compilers. The obvious application of a BNF parser is in writing translators for other languages. (This should certainly strenghten Forth's claim as a language to write other languages.)

Command interpreters. Complex applications may have an operator interface sufficiently complex to merit a BNF description. For example, this parser has been used in an experimental lighting control system; the command language occupied 30 screens of BNF.

Pattern recognition. Aho & Ullman [AH077] note that any construct which can be described by a regular expression, can also be described by a context-free grammar, and thus in BNF. [AH077] identifies some uses of regular expressions for pattern recognition problems; such problems could also be addressed by this parser.

An extension of these parsing techniques has been used to impement a Snobol4-style pattern matcher [ROD89a].

Goal directed evaluation. The process of searching the parse tree for a successful result is essentially one of "goal-directed evaluation." Many problems can be solved by goal-directed techniques.

For example, a variation of this parser has been used to construct an expert system [ROD89b].

10. References

[AH077] Alfred Aho and Jeffrey Ullman, Principles of Compiler Design, Addison-Wesley, Reading, MA (1977), 604 pp.

[ROD89a] B. Rodriguez, "Pattern Matching in Forth," presented at the 1989 FORML Conference, 14 pp.

Program Listing

```
                                                           Scr #      3
\ BNF Parser                                (c) 1988 B. J. Rodriguez
0 VARIABLE SUCCESS
: <BNF   SUCCESS @ IF  R> IN @ >R DP @ >R  >R
   ELSE  R> DROP  THEN ;
: BNF>   SUCCESS @ IF  R>  R> R> 2DROP   >R
   ELSE  R>  R> DP ! R> IN !  >R THEN ;
: |    SUCCESS @ IF  R> R> R> 2DROP DROP
   ELSE  R> R> R> 2DUP >R >R IN ! DP !  1 SUCCESS !  >R THEN ;
: BNF:   [COMPILE] : SMUDGE COMPILE <BNF ; IMMEDIATE
: ;BNF   COMPILE BNF> SMUDGE [COMPILE] ; ; IMMEDIATE

: @TOKEN ( - n)   IN @ TIB @ + C@ ;
: +TOKEN ( f)    IF 1 IN +! THEN ;
: =TOKEN ( n)    SUCCESS @ IF @TOKEN =  DUP SUCCESS ! +TOKEN
   ELSE DROP THEN ;
: TOKEN ( n)    <BUILDS C, DOES> ( a)  C@ =TOKEN ;

                                                           Scr#       4
\ BNF Parser - 8086 assembler version       (c) 1988 B. J. Rodriguez
0 VARIABLE SUCCESS
CODE <BNF    -1 # SUCCESS #) TEST, NE IF,      \ if passing,
      4 # RP SUB,    0FDFE # W MOV, ( U ptr)   \     checkpoint
      ' IN @ [W]  AX MOV, AX 2 [RP] MOV,       \     and continue
      ' DP @ [W]  AX MOV, AX 0 [RP] MOV,
   ELSE, 0 [RP] IP MOV,  RP INC,  RP INC,      \  else, exit now!
   THEN, NEXT

CODE BNF>    -1 # SUCCESS #) TEST, EQ IF,      \  if failing,
       0FDFE # W MOV, ( U ptr)                  \     backtrack to
       0 [RP] AX MOV, AX ' DP @ [W] MOV,        \     checkpoint
       2 [RP] AX MOV, AX ' IN @ [W] MOV,
    THEN, 4 # RP ADD, NEXT                      \  discard checkpoint
                                                \     and continue


                                                           Scr#       5
\ BNF Parser - 8086 assembler version       (c) 1988 B. J. Rodriguez
CODE |   -1 # SUCCESS #) TEST, NE IF,        \ if passing,
      4 # RP ADD,                            \   discard checkpoint
      0 [RP] IP MOV, RP INC, RP INC,         \   and exit now
   ELSE, 0FDFE # W MOV,                      \ else, backtrack,
      0 [RP] AX MOV,    AX ' DP @ [W] MOV,   \   leaving checkpoint
      2 [RP] AX MOV,    AX ' IN @ [W] MOV,   \   stacked, and
      SUCCESS #) INC,                        \   set true for next
   THEN, NEXT                                \   alternate
```

```
                                                           Scr #       6
\ BNF Parser Example #1 - pattern recog.            18 9 88 bjr 19:41
\ from Aho & Ullman, Principles of Compiler Design, p.137
\ this grammar recognizes strings having balanced parentheses

HEX    28 TOKEN '('      29 TOKEN ')'      0 TOKEN <EOL>

BNF: <CHAR>     @TOKEN DUP 2A 7F WITHIN SWAP 1 27 WITHIN OR
   DUP SUCCESS ! +TOKEN ;BNF

BNF: <S>       '(' <S> ')' <S>   |   <CHAR> <S>   |   ;BNF

: PARSE     1 SUCCESS !    <S> <EOL>
  CR SUCCESS @ IF ." Successful " ELSE ." Failed " THEN ;



                                                           Scr#        7
\  BNF Parser Example    #2  - infix notation        18 9 88 bjr 14:54
HEX    2B TOKEN   '+'    2D  TOKEN '-'     2A  TOKEN  '*'     2F TOKEN '/'
       28 TOKEN   '('    29  TOKEN ')'     5E  TOKEN  '^'
       30 TOKEN   '0'    31  TOKEN '1'     32  TOKEN  '2'     33 TOKEN '3'
       34 TOKEN   '4'    35  TOKEN '5'     36  TOKEN  '6'     37 TOKEN '7'
       38 TOKEN   '8'    39  TOKEN '9'       0 TOKEN  <EOL>

BNF: <DIGIT>      '0'  | '1' | '2' |  '3' | '4' | '5' | '6' | '7'
    |  '8' | '9' ;BNF
BNF: <NUMBER>    <DIGIT> <NUMBER>    |     <DIGIT> ;BNF






                                                           Scr#        8
\ BNF Parser Example     #2 - infix notation         18 9 88 bjr 15:30
\ from Aho & Ullman,     Principles of Compiler Design, pp.135,178
: [HERE]     HERE 0 ,   -2 CSP +!  ;    IMMEDIATE

BNF:   <ELEMENT>     '(' [HERE]  ')'  |    <NUMBER> ;BNF
BNF:   <PRIMARY>     '-' <PRIMARY>    |   <ELEMENT> ;BNF
BNF:   <FACTOR>    <PRIMARY> '^' <FACTOR> | <PRIMARY> ;BNF
BNF:   <T'>     '*' <FACTOR> <T'> | '/' <FACTOR> <T'> ;BNF
BNF:   <TERM>    <FACTOR> <T'> ;BNF
BNF:   <E'>     '+' <TERM> <E'> | '-' <TERM> <E'>    ;BNF
BNF:  <EXPRESSION>      <TERM> <E'> ;BNF
' <EXPRESSION> CFA SWAP !      \ fix the recursion in <ELEMENT>

: PARSE     1 SUCCESS !     <EXPRESSION> <EOL>
   CR SUCCESS @ IF  ." Successful " ELSE ." Failed " THEN ;


                                                           Scr #       9
 \ BNF  Example #3      code generation               18 9 88 bjr 21:57
HEX    2B TOKEN   '+'    2D  TOKEN '-'     2A  TOKEN  '*'     2F TOKEN'/'
       28 TOKEN   '('    29  TOKEN ')'     5E  TOKEN  '^'
       30 TOKEN   '0'    31  TOKEN '1'     32  TOKEN  '2'     33 TOKEN '3'
       34 TOKEN   '4'    35  TOKEN '5'     36  TOKEN  '6'     37 TOKEN '7'
       38 TOKEN   '8'    39  TOKEN '9'       0 TOKEN  <EOL>

BNF: {DIGIT}      '0'  | '1' | '2' |  '3' | '4' | '5' | '6' | '7'
    |  '8' | '9' ;BNF
BNF: <DIGIT>      @TOKEN {DIGIT} C, ;BNF

```

```
BNF: <NUMBER>      <DIGIT> <NUMBER>    |    <DIGIT> ;BNF

: (,")    R COUNT DUP 1+ R> + >R       HERE SWAP DUP ALLOT CMOVE ;
: ,"    COMPILE (,") 22 WORD HERE       C@ 1+ ALLOT  ;    IMMEDIATE


                                                              Scr#       10
\  BNF Example #3       code generation               18 9 88 bjr 21:57
:  [HERE]     HERE 0  ,   -2 CSP +!  ;    IMMEDIATE

BNF: <ELEMENT>     '('   [HERE]  ')'
                |   <NUMBER> BL C, ;BNF
BNF: <PRIMARY>      '-'  <PRIMARY>  ," MINUS "
                |    <ELEMENT> ;BNF
BNF: <FACTOR>      <PRIMARY> '^' <FACTOR>      ," POWER "
                |  <PRIMARY> ;BNF
BNF: <T'>      '*' <FACTOR>     ," * "    <T'>
            |  '/' <FACTOR>     ," / "    <T'>
            |  ;BNF
BNF: <TERM>     <FACTOR> <T'>       ;BNF
BNF: <E'>      '+' <TERM>    ." + "   <E'>
            |  '-' <TERM>    ." - "   <E'>
            |  ;BNF

                                                              Scr#       11
 \ BNF Example #3 - code generation                   18 9 88 bjr 21:57
 BNF: <EXPRESSION>       <TERM> <E'> ;BNF
 ' <EXPRESSION> CFA SWAP             \ fix the recursion in <ELEMENT>

 : PARSE    HERE 1 SUCCESS !         <EXPRESSION> <EOL>
    CR SUCCESS @ IF HERE OVER - DUP MINUS ALLOT TYPE
    ELSE ." Failed" THEN ;
```