

Specification of Programming Language Syntax

by

Martin Richards

`mr10@cl.cam.ac.uk`

`http://www.cl.cam.ac.uk/users/mr10/`

Computer Laboratory

University of Cambridge

Revision date: Mon Nov 4 10:57:01 GMT 2024

Abstract

Normally Backus Naur Form (BNF) or extended BNF (EBNF) is used to specify the syntax of programming languages. Users typically find such specifications hard to interpret and it is often not easy to check whether a parser conforms precisely with the BNF specification. There is also the problem that the BNF may be ambiguous.

This document suggests way to specify the syntax of programming languages that is concise and easily understood by users. It essentially specifies a recursive descent parsing algorithm making it easy to implement a parser conforms precisely to the specification. Syntax specified in this way is guaranteed to be unambiguous.

Keywords: BNF, EBNF, transition diagrams, ambiguity.

Chapter 1

Introduction

Backus Naur Form (BNF) was first used in the Algol 60 Report to give a precise description of its syntax.

John Backus and Peter Naur were members of the committee that developed Algol 60.

Since then almost all programming languages have had their syntax specified using BNF or Extended BNF (EBNF). EBNF is a slightly more compact notation to specify a BNF grammar.

Algol 60 was a tour de force since at that time the most popular languages were COBOL, FORTRAN 66, LISP and various Autocodes.

1.1 Example BNF grammar for expressions

```

Bexp ::= name
       number
       ( E )
       + E
       - E

```

```

E  ::= Bexp
      E ^ E
      E * E
      E / E
      E + E
      E - E

```

An example mathematical expression

$$x * 2^{-n^2} - 16 / -2 * 3$$

can be written satisfying the above grammar as follows:

$$x * 2 ^ {-n^2} - 16 / -2 * 3$$

1.2 Syntax is difficult

At about the age of 7 we are taught that multiplication is more binding than addition. So $1 + 2 \times 3$ evaluates to 7 not 9.

I was told to remember the word BODMAS which stands for

Brackets Of Division Multiplication Addition and Subtraction

indicating that, for instance, multiplication is more binding than addition.

Unfortunately BODMAS is just wrong and misleading. For instance, what does it say about:

$$24 \times 6 \div 2 + 1 - 5 + 6$$

Or even:

$$x + y + z$$

Consider the following two assignments:

```
s = x + z;  
t = x + y + z;
```

```
Bexp := name  
      number  
      ( E0 )  
      + E1  
      - E1
```

```
E2 ::= Bexp  
      Bexp ^ E2
```

```
E1 ::= E2  
      E1 * E2  
      E1 / E2
```

```
E0 ::= E1  
      E0 + E1  
      E0 - E1
```

```
Prog ::= E0 <eof>
```

This grammar could be written using category names similar to those in the C++ grammar given in the ISO/IEC 14882:1998(E) specification

```
<basic expression> ::=
    <name>
    <number>
    ( <additive expression> )
    + <multiplicative expression>
    - <multiplicative expression>

<power expression> ::=
    <basic expression>
    <basic expression> ^ <power expression>

<multiplicative expression> ::=
    <power expressio>
    <multiplicative expression> * <power expression>
    <multiplicative expression> / <power expression>

<additive expression> ::=
    <multiplicative expression>
    <additive expression> + <multiplicative expression>
    <additive expression> - <multiplicative expression>

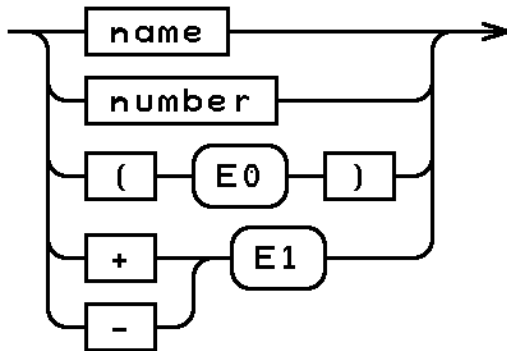
<program> ::=
    <additive expression> <eof>
```

This grammar has
6 syntactic categories, 14 productions and 10 terminal symbols.

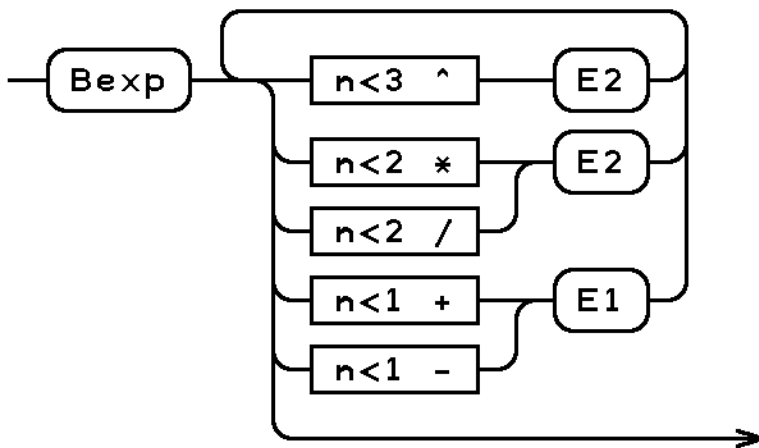
The ISO/IEC 14882:1998(E) grammar for C++ has
196 categories, 640 productions and 192 terminal symbols.

1.3 The transition diagrams

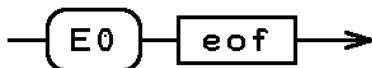
The definition of Bexp



A more compact definition of En

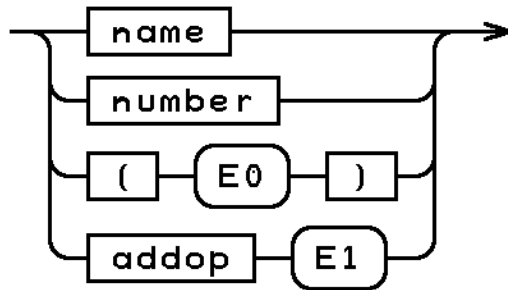


Definition of Prog

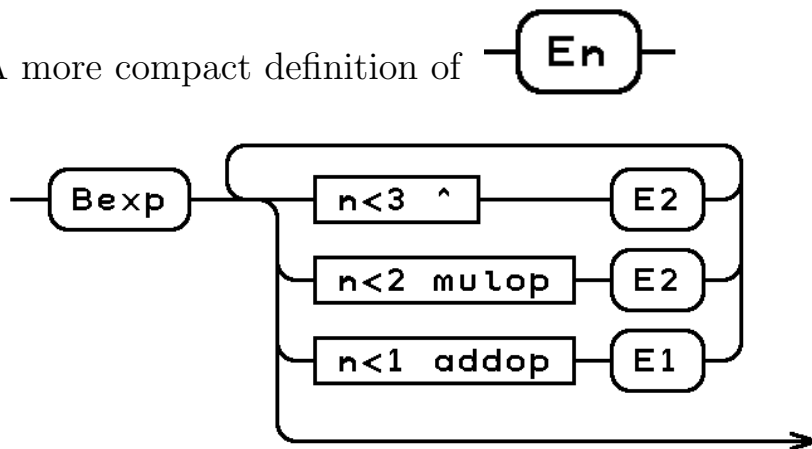


1.4 The compacted transition diagrams

A more compact definition of Bexp



A more compact definition of En



Definition of Prog



1.5 Implementing the Parser

```

LET rbexp() = VALOF
{ // Read a basic expression returning the corresponding parse tree
  LET op = token
  LET res = 0
  SWITCHON op INTO
  { DEFAULT:
    synerr("Bad token at the start of an expression")
    lex()
    RESULTIS 0

    CASE s_name:
    CASE s_number:
      res := mk2(op, lexval)
      lex()
      RESULTIS res

    CASE s_lparen:
      res := rnexp(0)
      checkfor(s_rparen, "'')' expected")
      RESULTIS res

    CASE s_add:
    CASE s_sub:
      res := rnexp(1)
      IF op=s_sub RESULTIS mk2(s_neg, res)
      RESULTIS res
  }
}

AND rnexp(n) = VALOF
{ lex()
  RESULTIS rexp(n)
}

```

```
AND rexp(n) = VALOF
{ LET res = rbexp()

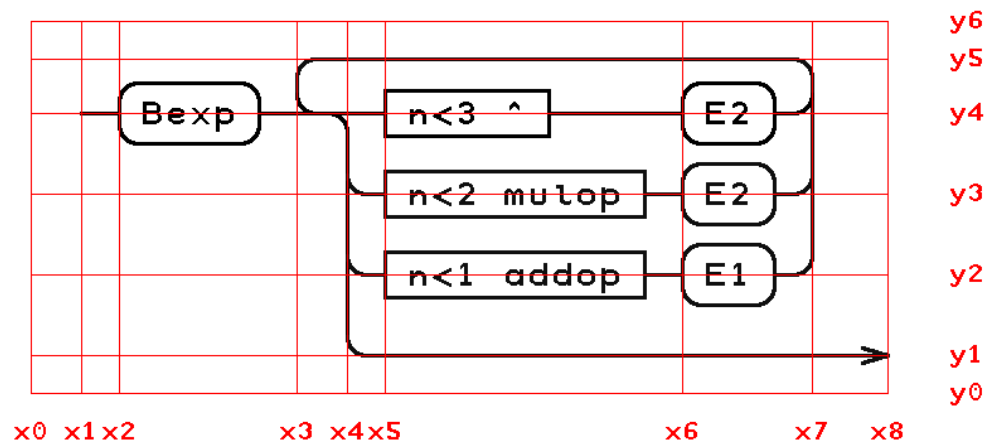
  { LET op = token
    SWITCHON op INTO
    { DEFAULT:
      RESULTIS res

      CASE s_power:
        UNLESS n<3 RESULTIS res
        res := mk3(op, res, rnexp(2))
        LOOP

      CASE s_mul:
      CASE s_div:
        UNLESS n<2 RESULTIS res
        res := mk3(op, res, rnexp(2))
        LOOP

      CASE s_add:
      CASE s_sub:
        UNLESS n<1 RESULTIS res
        res := mk3(op, res, rnexp(1))
        LOOP
    }
  } REPEAT
  RESULTIS res
}
```

1.6 How to draw the flow graphs



Code to draw the items at level y_4

```
drawcatboxL (y4, x1, x2, x5, "Bexp")
drawtestboxL(y4, x5, x5, x6, "n<3 ^")
drawcatboxL (y4, x6, x6, x7-r, "E2")
rndcorner(2, x3, y4, r)
rndcorner(0, x4, y4, r)
rndcorner(3, x7, y4, r)
moveto(x4, y4-r)
drawto(x4, y1+r)
```

Section Syntactic ambiguity

The rules associated with the flow graphs are:

- 1) Backtracking through a test box that matches a lexical token is not permitted.
- 2) At a path division point the left hand branch is always tried first.

The rules ensure the grammar specified by the flow graphs contains no ambiguities.

It is easy to create flow graphs in which every loop will match at least one lexical, and it is easy to check that this property holds.

Note that BNF grammars can contain ambiguities.

1.7 BNF ambiguities

From the Web BNF I have obtained specifications for the languages C, C++ and Java. All three essential the following productions.

`|selection-statementi ::= if (|expressioni) |statementi`

`|selection-statementi ::= if (|expressioni) |statementi else |statementi`

`|statementi ::= |selection-statementi`

This contains an ambiguity since there are two way to parse the following

```
res = 0;
if (x>0) if (y>0) res=2; else res = 1;
```

If x is zero this code will set `res` to either zero or one depending on how the author of the compiler interpreted the grammar.

This is exactly the same ambiguity that was present in the Algol 60 Report 64 years ago.

My conclusion is that language designers and/or people who write programming language manuals choose to specify the grammar using BNF but don't notice or care if the BNF specification is ambiguous.

Chapter 2

Syntax Diagrams for BCPL

This appendix gives the precise syntax of BCPL as it is now, at least in February 2022. It includes the floating point operators, the FLT feature and the newly added pattern matching constructs. It also contains some constructs from older versions of BCPL to make compilation of older BCPL programs easier.

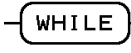

The syntax of programming languages is often specified using Backus Naur Form or BNF. Mathematicians like BNF notation because of its simplicity, power and interesting properties, while language designers like it because the rules just confirm their understanding of the language grammar they are designing. For users, understanding a grammar from its BNF specification is harder. There are typically a hundred or more of syntactic categories, many with artificial names, and a greater number of rules. Understanding the rules is hard because they mostly depend on each other. There is also sometimes a problem noticing whether a BNF grammar is ambiguous. Indeed it is not possible, in general, to write a program that can determine whether a BNF grammar is ambiguous, and it is also not always easy to write a parser that precisely agrees with the BNF specification.

Even though much research has been done in this area resulting in packages such as Lex and Yacc, I have decided to specify the grammar of BCPL using a method based on transition diagrams. This method gives a precise specification of the parsing algorithm. The diagrams are easy to understand and have the advantage that the grammar is unambiguous. It is also easy to check that the parser in a compiler conforms precisely with this specification. These diagrams can also be used as the basis of a program to find a minimum cost syntactic repairs, resulting in better error messages. Such a program, `checksyntax.b`, is currently under development and is available in the BCPL distribution.

The BCPL syntax is given using the diagrams shown in figures 2.1, 2.2, 2.3, 2.4, 2.5, 2.6 and 2.7 for the syntactic categories `Prog`, `D`, `Mlist`, `Pn`, `C`, `Bexp` and `En`. In the diagrams these categories are represented by the rounded boxes:

 `Prog`, `D`, `Mlist`, `Pn`, `C`, `Bexp`, and `En`.

A rectangular box is called a test box and may contain a terminal symbol as in

 or , or a label representing a set of terminal symbols or some other condition. These test box labels are specified in the following table.

Label	Possible symbols or condition
name	A name not preceded by FLT
fname	A name possibly preceded by FLT
number	Integer or floating point constant
const	Integer or floating point constant, BITSPERBCPLWORD character constant, TRUE, FALSE or ?
bpat	Possibly signed integer or floating point constant, character constant, TRUE, FALSE, ?, or a name not preceded by FLT
string	A string constant
mulop	* / MOD #* #/ #MOD
posop	+ - ABS #+ #- #ABS
addop	+ - #+ #-
relop	= ~= < <= > >= #=# ~= #< #<= #> #>=
fcond	-> #->
range	.. #..
jcom	NEXT EXIT BREAK LOOP ENDCASE RETURN
assop	:= *= /= MOD:= += -= #:= #*:= #/:= #MOD:= #+:= #-:= <<:= >>:= &:= = EQV:= XOR:=
iscall	This is only satisfied if the most recent construct was a function, routine or method call
isname	This is only satisfied if the most recent construct was a name nor enclosed in parentheses
nonl	This is only satisfied if the previous and current tokens are on the same line
defop	This is satisfied when reading a GLOBAL declaration if the current token is : This is also satisfied when reading a MANIFEST or STATIC declaration if the current token is =
eof	This is only satisfied if the program file is exhausted

A test box that specifies one or more terminal symbols can only be traversed if it matches the current symbol in the program. Some test boxes have side conditions such as `n<5` which must also be satisfied. When a box successfully matches a terminal symbol, input is advanced to the next symbol of the program.

Test and category boxes are connected by paths which may contain branch points where paths diverge, and join points where paths converge. Each diagram has an entry point and an exit point, and every path in it has an implied direction.

The diagrams specify an infinite extended flow graph obtained by starting with the category **Prog** and repeatedly replacing category boxes by their definitions, substituting the parameter **n** where necessary. Every test box in the extended graph having a side condition will now compare two explicit integers and so is either equivalent to a test box without a side condition if the comparison is successful, or the box is eliminated if the condition fails. The extended graph thus only contains test boxes without side conditions.

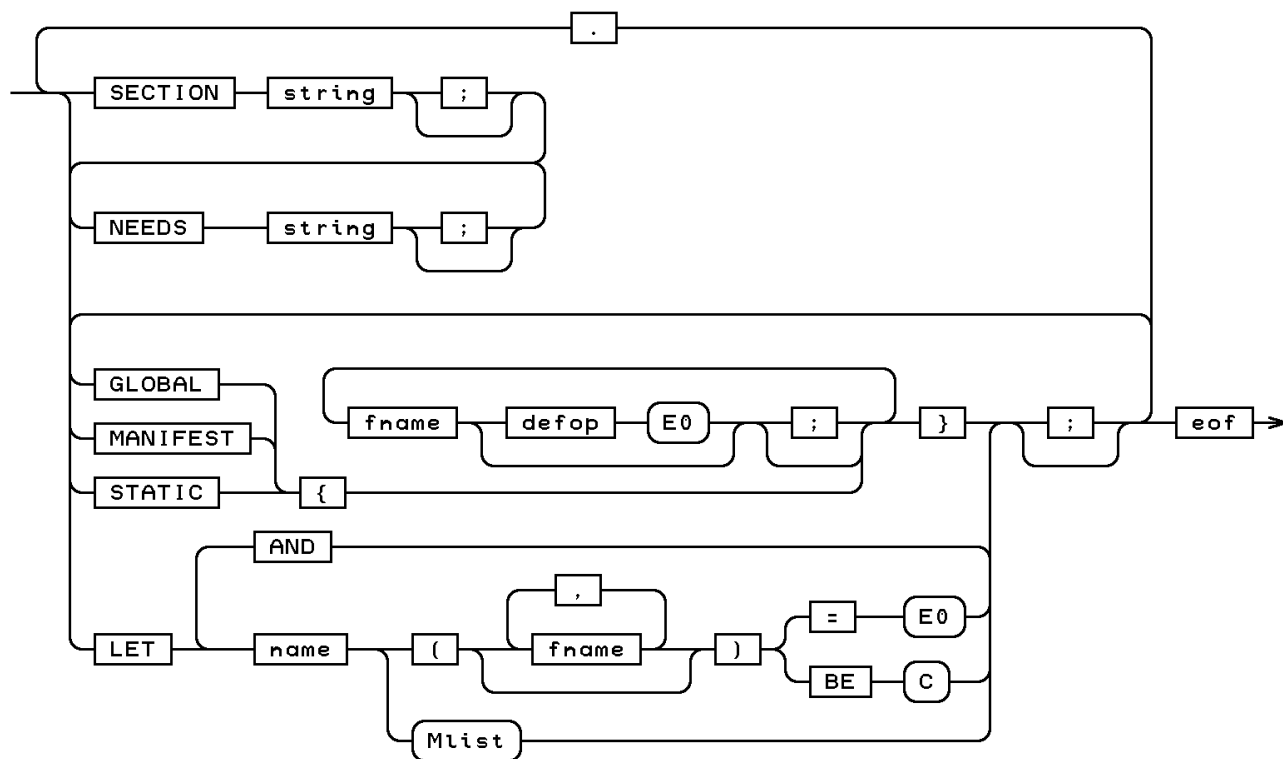
The parsing algorithm searches through the extended flow graph trying to find a path containing a sequence of test boxes that match the terminal symbols of the program being parsed. Whenever a branch point is encountered, the left branch is tested first, only trying the right branch when all test boxes reachable from the left branch fail. If a test box is successful and all boxes reachable from it fail, the program is syntactically incorrect. A program is only syntactically correct if the exit point of the extended graph can be reached.

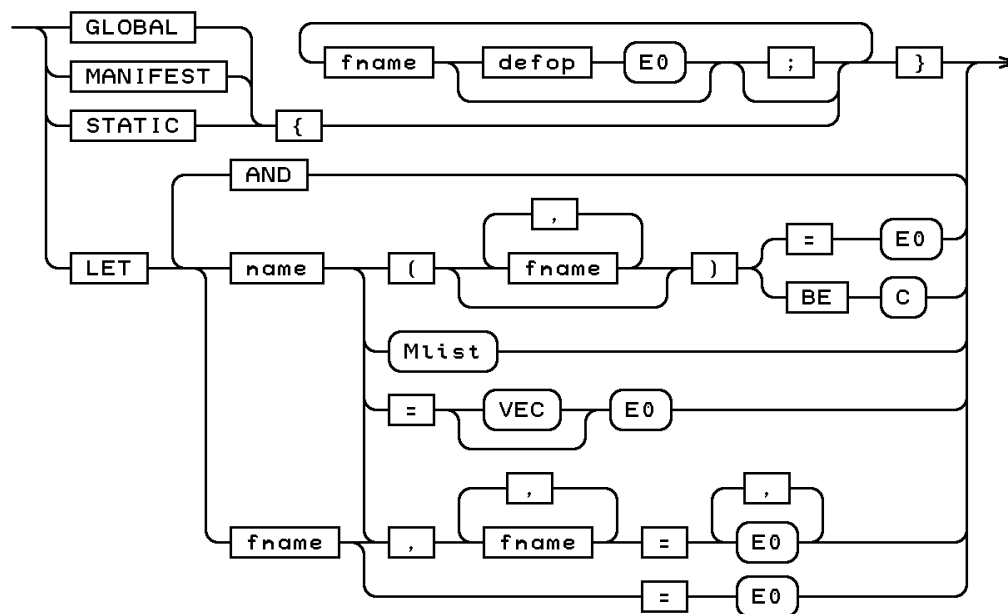
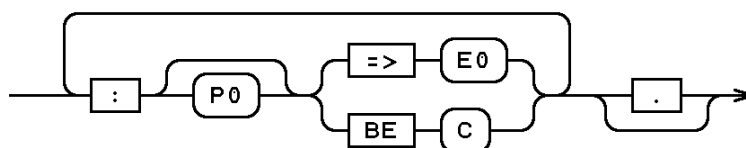
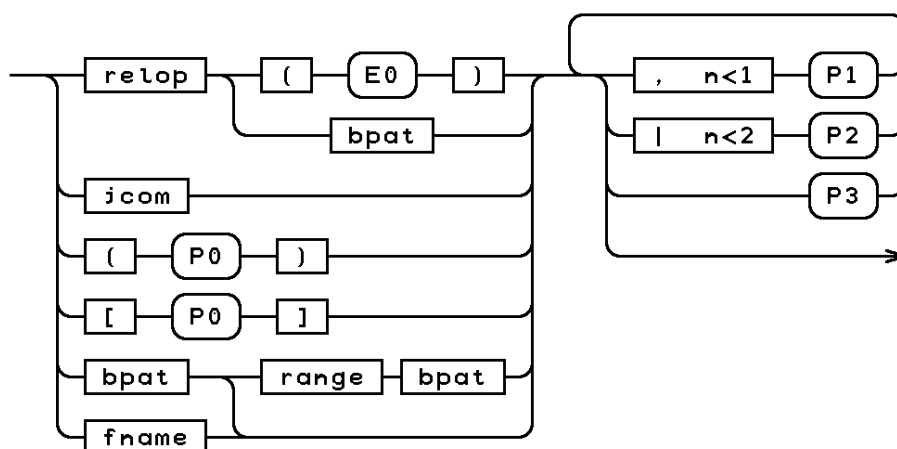
To keep the diagrams as simple as possible there are some syntactic constraints they do not cover. These are as follows.

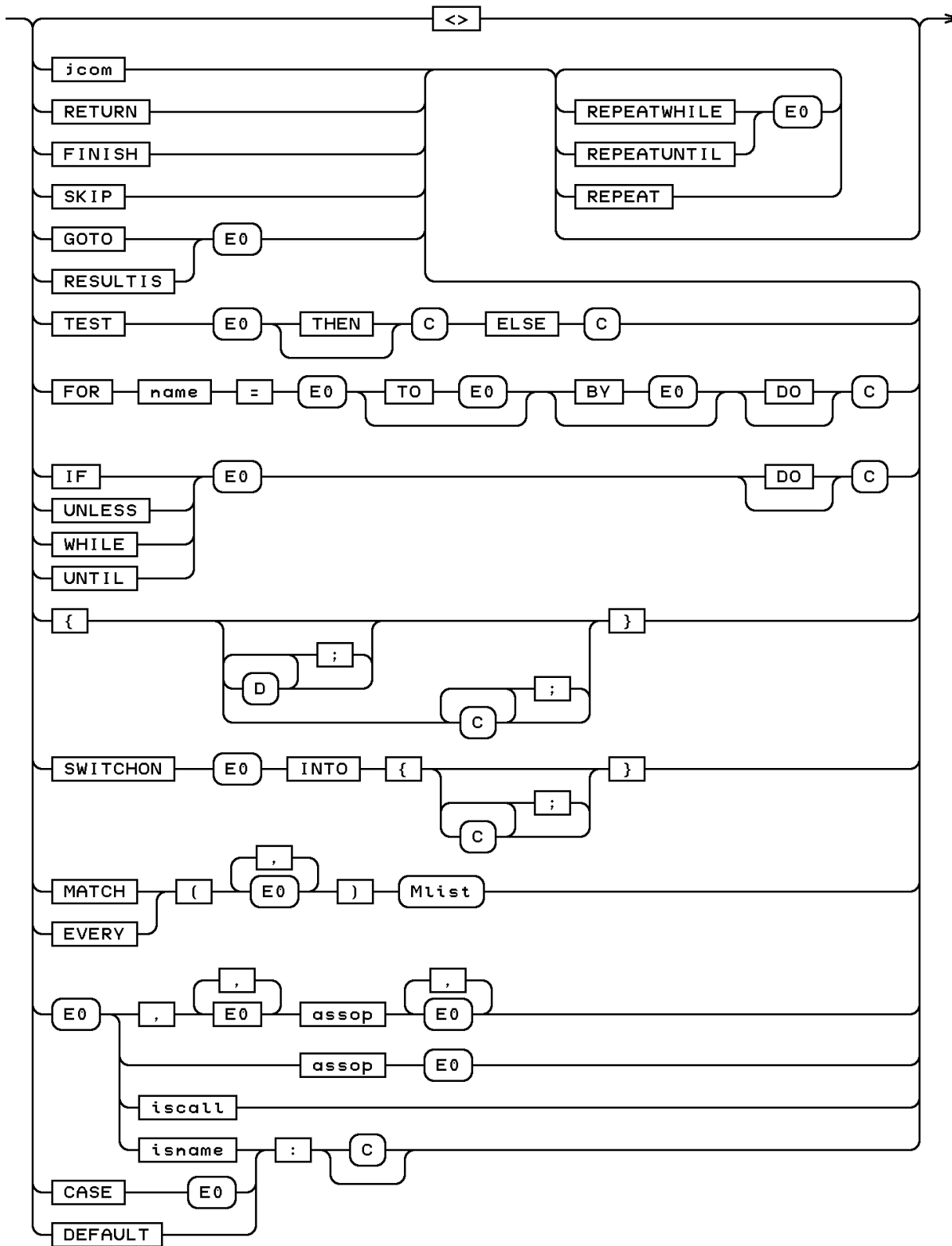
- 1) Names declared in **GLOBAL** declarations must use the defining operator :
- 2) Names declared in **MANIFEST** and **STATIC** declarations must use the defining operator =
- 3) In a match list the defining operator namely => or BE must be the same in each match item.
- 4) In a **MATCH** or **EVERY** expression, the defining operator in the match items must all be =>
- 5) In a **MATCH** or **EVERY** command, the defining operator in the match items must all be BE
- 6) The operands of .. and #.. must be either be names or manifest constant expressions.
- 7) The number of patterns separated by commas in square brackets must not exceed 255.
- 8) The depth of nesting of square brackets in patterns must not exceed 4.
- 9) In a local variable declaration, the number of names must equal the number of initial value expressions.
- 10) In assignment commands, the number of expressions on the left and right sides must be the same.
- 11) **FLT** can only precede names when the name is being declared as local variable, a formal parameter or a pattern variable.
Note that **FOR** loop control variables may not be prefixed by **FLT**

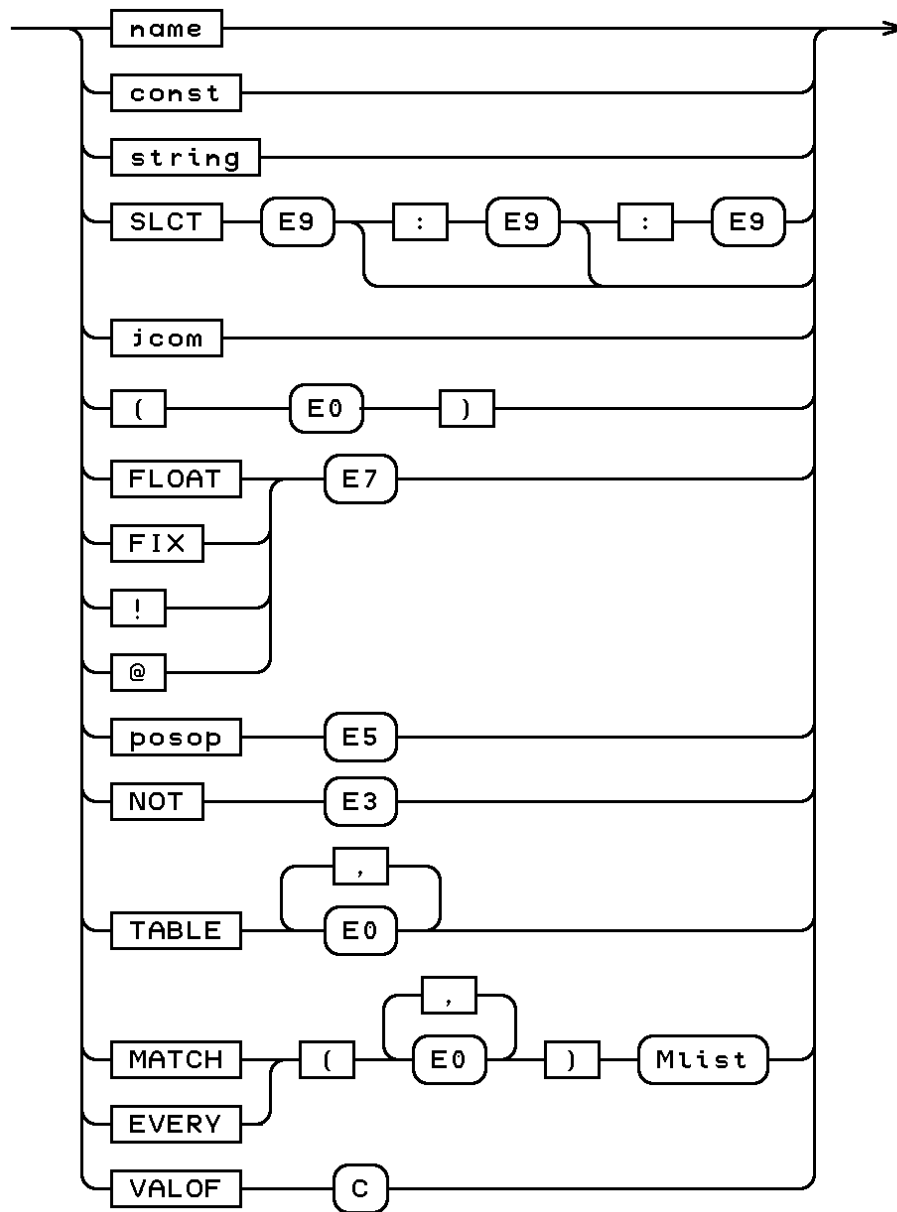
For compatibility with older versions of BCPL some terminal symbols have synonyms as follow.

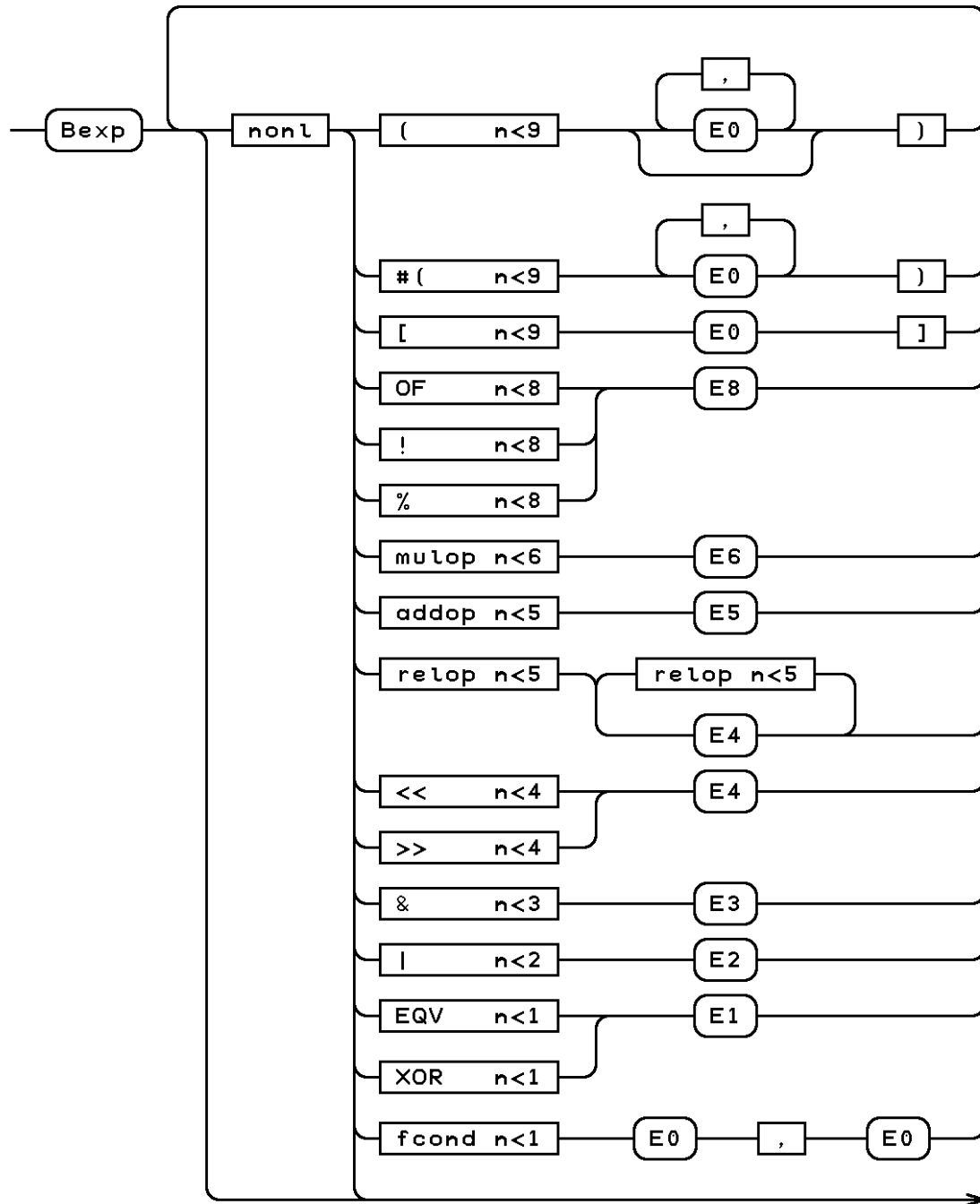
Symbol	Possible synonyms
{	\$(, possibly tagged
}	\$(, possibly tagged
DO	THEN
THEN	DO
MOD	REM
NOT	~
OF	::
= ~=	EQ NE
< <=	LS LE
> >=	GR GE
<< >>	LSHIFT RSHIFT
&	LOGAND LOGOR
XOR	NEQV

Figure 2.1: The definition of **Prog**

Figure 2.2: The definition of $-D-$ Figure 2.3: The definition of $-Mlist-$ Figure 2.4: The definition of $-P_n-$

Figure 2.5: The definition of C

Figure 2.6: The definition of $Bexp$

Figure 2.7: The definition of `En`