



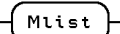



Appendix A



BCPL Syntax Diagrams

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 some 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 unambiguous. 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 the compiler conforms precisely with this specification.

The BCPL syntax is given using the diagrams shown in figures A.1, A.2, A.3, A.4, A.5 and A.6 for the syntactic categories **Prog**, **D**, **Mlist**, **Pn**, **C** and **En**. In the diagrams these categories are represented by the rounded boxes: , , , ,  and , respectively.

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
number	Integer or floating point constant
const	Integer or floating point constant, character constant, TRUE , FALSE or ?
bpat	Possibly signed integer or floating point constant, character constant, TRUE , FALSE , ?, or a name possibly preceded by FLT
name	A name not preceded by FLT
fname	A name possibly preceded by FLT
mulop	* / MOD #* #/ #MOD
posop	+ - ABS #+ #- #ABS
addop	+ - #+ #-
relop	= ~= < <= > >= #=# ~= #< #<= #> #>=
cond	-> #->
range	.. #..
jcom	NEXT EXIT BREAK LOOP ENDCASE
assign	:= *:= /:= 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
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

Test and category boxes are connected by paths which may contain branch points where paths diverge to the left or right, 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  and repeatedly replacing category boxes by their defined flow graphs, substituting the parameter **n** where necessary. The extended graph can be thought of as only containing test boxes.

A test box can only be traversed if the condition or the possible terminal symbols it specifies match the current program input. If the box successfully matches a terminal symbol, input is advanced to the next symbol of the program.

A test box can contain a side condition such as **n**<5, and in the extended flow graph **n** will have always been replaced by an explicit integer. Such a box can only be traversed if the condition is satisfied.

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. But whenever the algorithm encounters a branch point, it

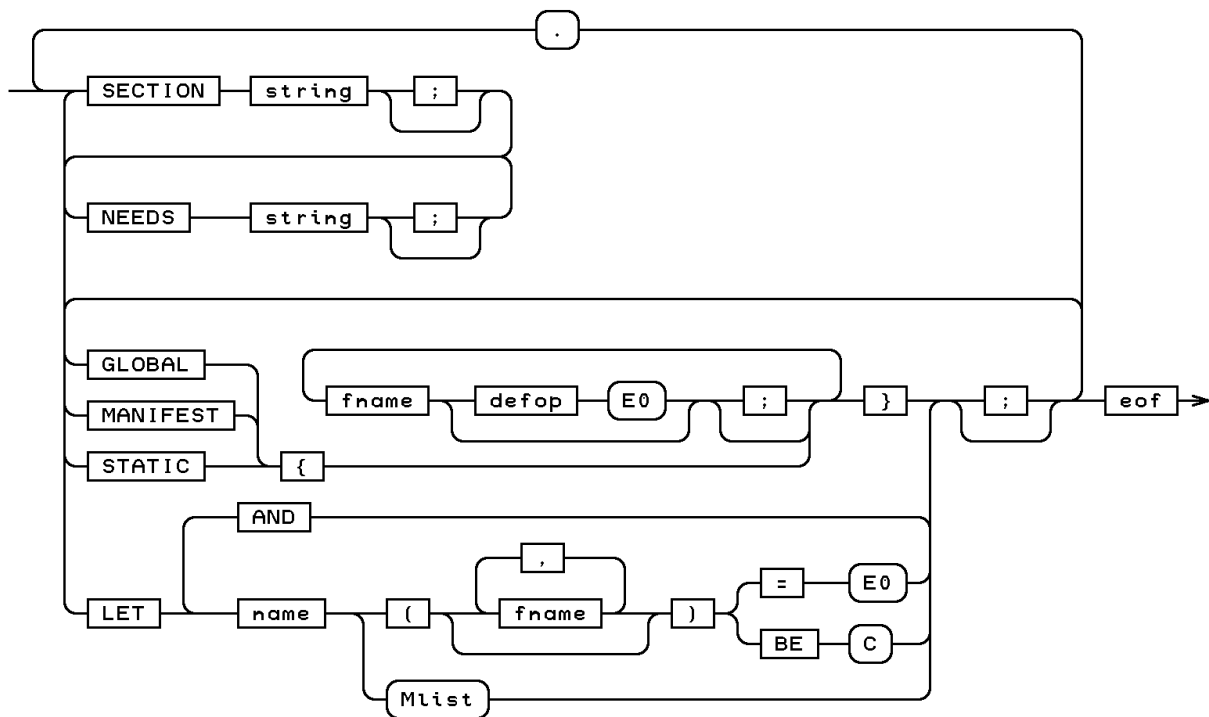
always tries the left branch first. If a test box is satisfied and all boxes reachable from it fail, the program is syntactically incorrect. If the exit point of the extended flow graph is reached, the program is syntactically correct.

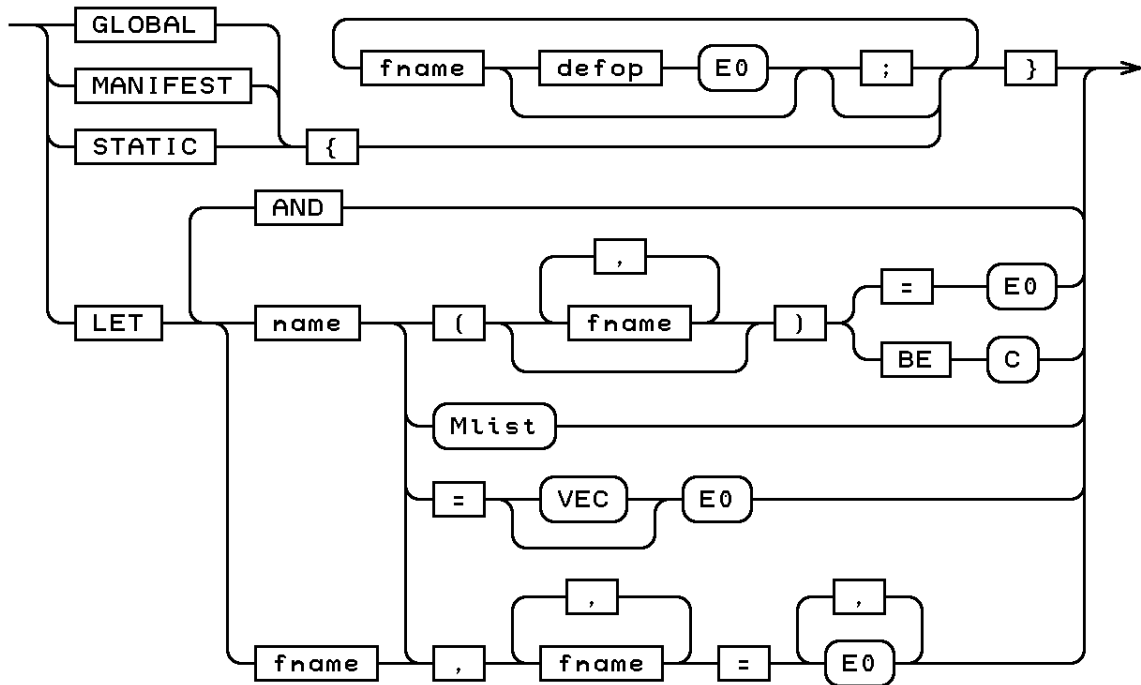
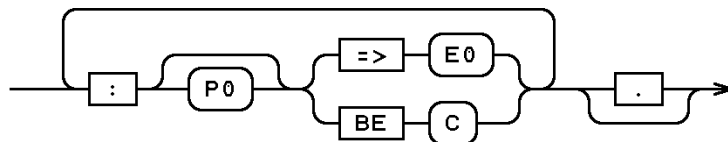
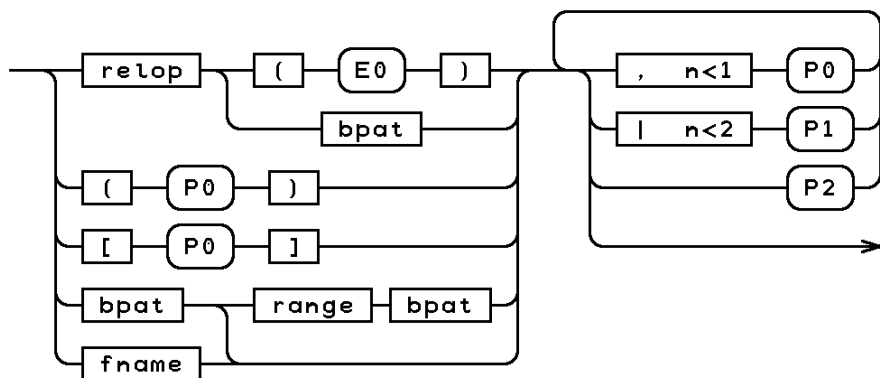
To keep the diagrams as simple as possible there are some syntactic constraints they do not cover. These are as follows.

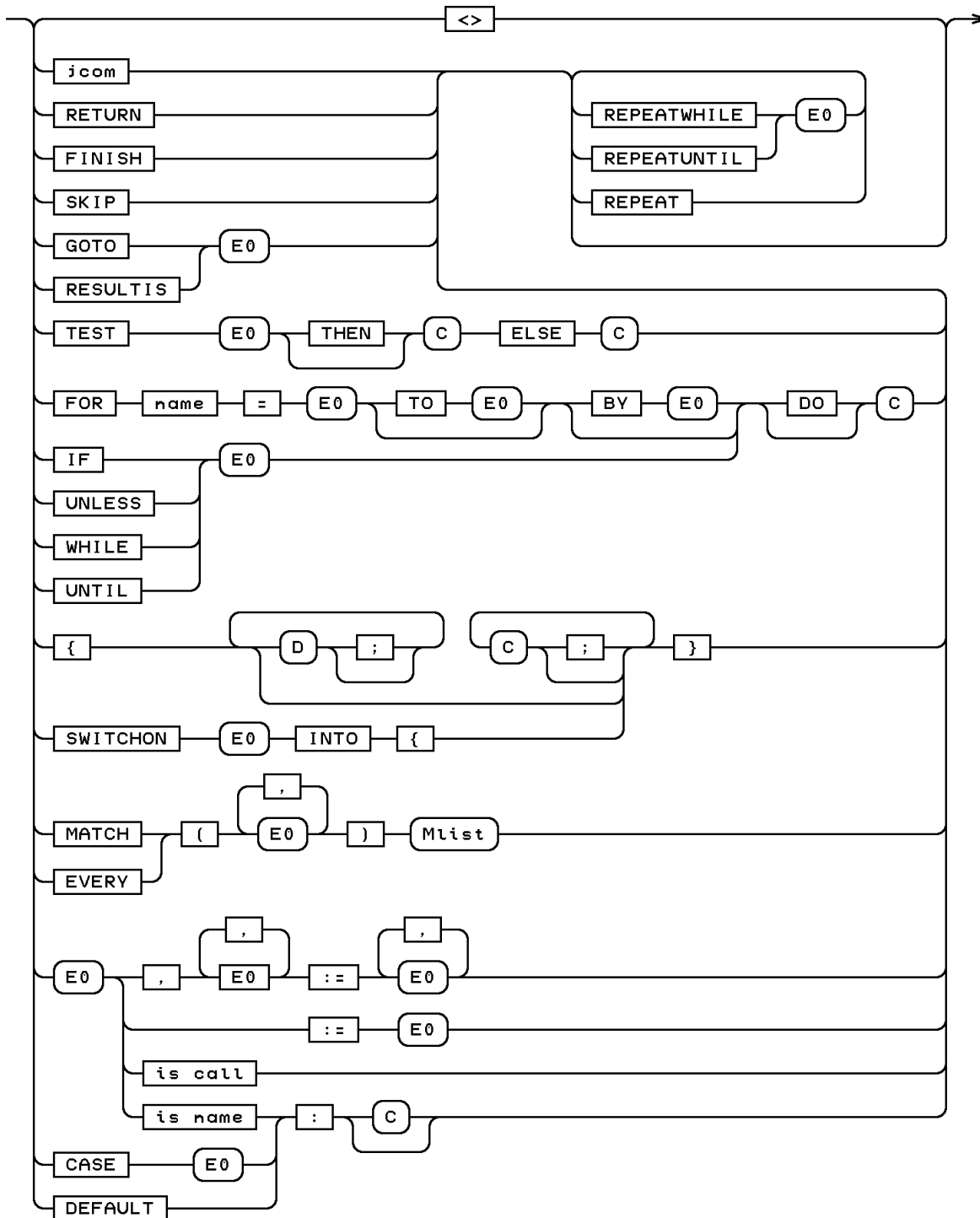
- 1) Names declared in **GLOBAL** declarations must use the defining operator :
- 2) Names declared in **MANIFEST** and **STATIC** declarations must use the defining operator =
- 3) In a match list the defining operator namely => or BE must be the same in each match item.
- 4) In a **MATCH** or **EVERY** 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 range must be 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.

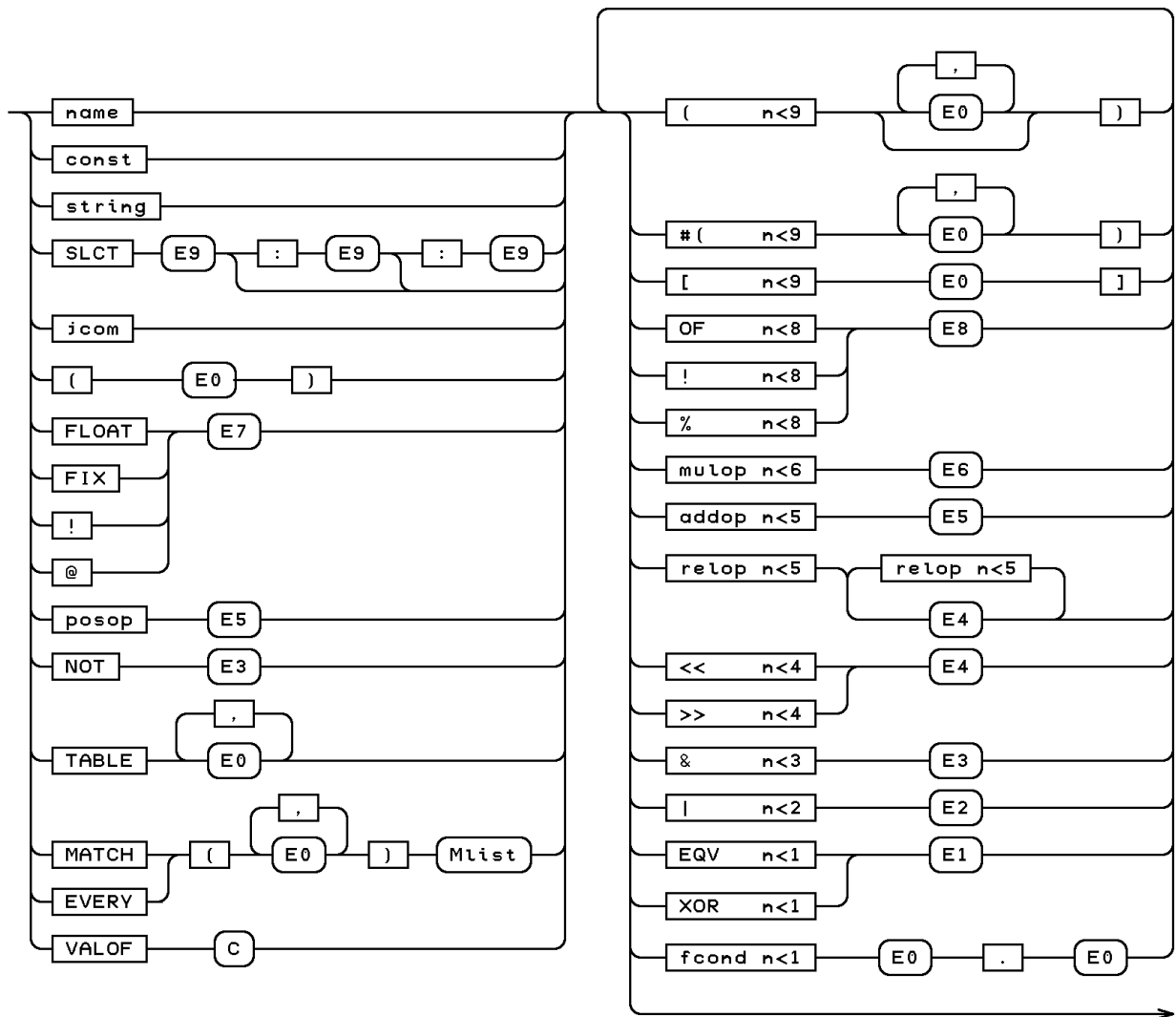
For compatibility with older versions of BCPL some terminal symbols have synonyms as follow.

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 A.1: The definition of `Program`

Figure A.2: The definition of $-D-$ Figure A.3: The definition of $-Mlist-$ Figure A.4: The definition of $-P_n-$

Figure A.5: The definition of $\langle C \rangle$

Figure A.6: The definition of $-E_n-$

