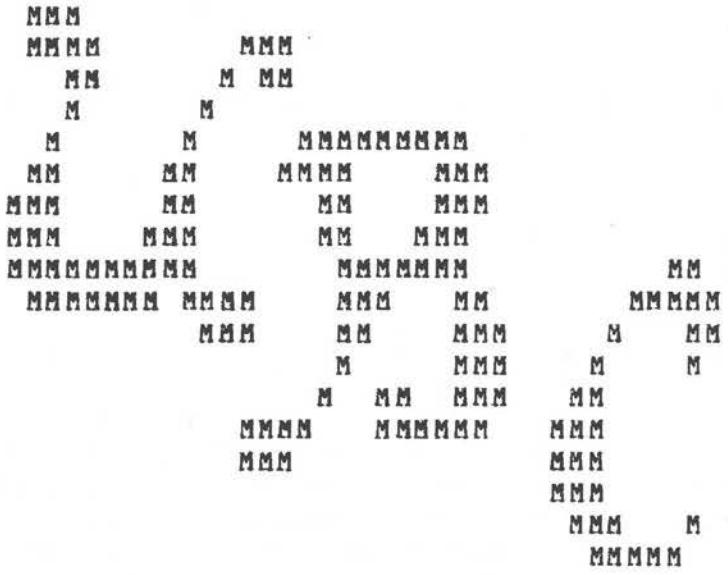

An Unambiquous Scanner for Special Character Tokens

by

R. A. Fraley

Technical Report 77-5

June 1977

Department of Computer Science
University of British Columbia
Vancouver, B. C.

An Unambiguous Scanner for Special Character Tokens

R. A. Fraley
University of British Columbia
June 22, 1977

Abstract

A fast algorithm for a general purpose scanner is presented. It includes a mechanism for permitting user-defined special character tokens. The scanner is able to separate strings of special characters without imposing arbitrary spacing rules on the programmer. An analysis shows that most special character tokens from selected languages could be handled properly by the scanner, even if they were in the same language. Many of the omitted tokens could be confused for combinations of operators, demonstrating the utility of the scanner for preventing lexical ambiguity. The special character analysis is extended to other classes of tokens.

An Unambiguous Scanner for Special Character Tokens

Input scanners for programming languages have been around for quite some time [7,14]. They divide the input string into units called "tokens". The normal implementation uses a finite-state machine to recognize the tokens. Special character operator tokens, such as ":=" and "**", are generally built into the scanner. Many scanners also contain a table of reserved words for the language; this feature shall not be discussed. The scanner described below has been developed as part of the Unlanguage project.

Traditional scanners are not very convenient for languages which change during compilation. The programmer is generally restricted to tokens which have been built into the scanner. Operators could, however, be defined in a symbol table, as is traditionally done with variables. Some languages, such as Snobol [8] and Algol 68 [21], already have a limited form of this capability. The defined operators must be chosen from a specified collection of symbols (when a word symbol is not used). The language Mary [4] allows symbols to be constructed as sequences of specified characters. The Unlanguage scanner also allows for the creation of new token symbols, but uses a different scanning technique than that used for Mary.

In designing such a scanner, we have two conflicting goals: efficiency and flexibility. Efficiency is important due to the number of characters processed by the scanner during a compilation, and flexibility is important so that the scanner doesn't place undue restrictions on the language. Consider the Pascal [13] statement

A(.I.)↑:=-B

If our scanner was being used with Pascal, it would need to produce the tokens ".)", "↑", ":=", and "-", among others. But suppose that our scanner isn't designed for a specific language. How will it know where one token ends and the next begins? We could require the programmer to put spaces between the tokens (the "SNOBOL technique").

A (. I .) ↑ := - B

This imposes unreasonable requirements on the programmer (and hence the language). The requirement is unreasonable because it differs from our normal use of symbols.

Another separation technique is to look up each potential token in a table. That is, we add one character at a time to the token until the resulting string is neither a token nor the initial portion of a token. We use the longest token matched by

the input, saving extra characters for future tokens (the "Mary technique"). This method is quite flexible, but the table look-up slows down the scanning process, and imposes some restrictions on the symbol table organization. The coding is complicated by the necessary back-up.

Lexical Ambiguity

A general scanner should face the issue of lexical ambiguity. This is a phenomenon which is found in a number of existing languages. FORTRAN [10] is a prime example of this phenomenon. Consider the expression fragments "12.EQ." and "12.E2". The first fragment has two tokens; the second has only one. "12" may be a token, but we must look past the "." and the "E" to the next character to find out. Our general scanner should never look ahead more than one character.

Another type of lexical ambiguity is found in the language Simula 67 [5]. Simula uses the operator ":-" for the assignment of reference values, as opposed to ":=" which is used for the assignment of data values. This new operator provides a problem for the scanner because the construct "[-10:-1]" is valid for the specification of array bounds. In this case ":-" is two tokens, ":" and "-". A specially designed scanner will produce two tokens when ":-" occurs inside of brackets, where a pointer assignment cannot occur. But if the scanner allows new operator definitions, a complex language analysis would be required to discover such facts. It would be desirable to eliminate this type of dependence on a specialized grammar analyzer.

One more type of lexical ambiguity occurs in the language C [18]. The operator "--" is used as a prefix operator. But "-" is also a prefix operator in C, so the expression "--B" could be interpreted as either "-- B" or as "-(-B)". We would have a similar problem if "--" was an infix operator: does "A--B" mean "A -- B" or "A - (-B)"? One could argue that the second interpretation is not very useful. But in a general scanner, we cannot know which combinations of operators are useful. We must prevent the unsuspecting user from creating such operators. For example, the user might define an assignment operator "<-" on a machine which has no left arrow. Some Boolean expressions may accidentally use this operator, as in "if A<-B then...". Here, separate "<" and "-" tokens were probably intended, although assignment may have been desired. The scanner described below will not recognize the "<-" as a token. The parser could accept the pair of tokens "<" followed by "-" if the language designer feels that the construct is safe, although the parser generator being developed for use with this scanner would issue a warning message.

Some scanners ignore the issue of lexical ambiguity. The "Mary technique" requires the programmer to separate tokens when an ambiguity arises. Some C compilers warn the user when a

construct might be ambiguous. Other languages, such as Algol 68, avoid lexical ambiguity by the choice of special character tokens. For the Unlanguage processor, special character tokens may be used as user-defined operators or delimiters. Lexical ambiguity is avoided as a protection for the user.

One way to prevent lexical ambiguity is to regard each special character as a complete token. This leaves the issue of ambiguity for the parser to worry about. The Unlanguage scanner can operate this way, but allowing the language designer the use of tokens seems to simplify his task. The technique used to eliminate lexical ambiguity of operators requires little extra work for the scanner with only minor limitations on the use of tokens in the language.

Token Classification

If one examines the use of special characters in existing languages, it becomes apparent that lexical ambiguity is avoided by the legal placement of tokens in the language. Special character symbols usually appear as prefix, infix, or postfix operators, which obey certain rules:

- -- Infix operators never appear together.
- -- A prefix operator is never followed by a postfix operator.
- -- An infix operator cannot follow a prefix operator or precede a postfix operator.
- -- A postfix operator may be followed by an infix operator, and an infix operator by a prefix operator.
- -- Any number of prefix operators may appear together, and any number of postfix opreators may appear together.
- -- Operands may be preceded by an infix or prefix operator, and followed by an infix or postfix operator.

Thus, for example, the infix operators ":=", ":", and "=" may all be present in an unambiguous language, because ":" and "=" as infix operators may never be adjacent. But if '=' is also a valid prefix operator, then ":" could be followed by a prefix "=", so that ":=" is ambiguous.

An operator token is a "prefix token", "infix token", or "postfix token" depending on the type of operator which it represents. If a token is used as both a prefix (postfix) and infix operator, it is a prefix (postfix) token. If it is both a prefix and a postfix operator, it is called a "bifix token".

The rules which we will develop for bifix tokens also apply to special character tokens used as operands.

Character Classification

Let us investigate the properties of the token classes based on the characters used to represent the tokens. If we are scanning an infix operator, we must gather together the one or more characters used to represent the operator. But the infix operator might be followed by a prefix operator. We must stop gathering up characters when the first character of the prefix operator is encountered. Likewise, when reading a prefix token, the scanner gathers up characters until the first character of the next prefix operator is found. (Of course, the scan also stops if a blank, letter, or digit is found.) This behavior suggests that the first character of a prefix operator must be chosen from a distinguished collection of characters.

Let's see some examples of this rule. For our examples, the characters "+", "-", "¬", and "(" will be the prefix characters. The expression "A:=-B" is broken into four tokens: "A", ":=", "-", and "B". Because the "-" symbol is one of the distinguished prefix characters, two tokens ":=" and "-" were formed instead of a single token ":=-". Likewise, because "=" is not a distinguished character, ":=" remained as a single token. The expression "A:=:B" has a single operator, since ":" is not distinguished. The expression "A+:=B" is valid even though "+" is distinguished: an infix operator may begin with a prefix character; a prefix operator must begin with such a character. The prefix characters may not appear after the first character of an operator, so tokens such as "++" and "$(" are not possible for this scanner.

Postfix tokens are the reverse of prefix tokens. It is the last character of a postfix token which is important. This postfix character signifies the end of the token, just as the prefix character signifies the beginning of a new token. Thus if ")" is a postfix character, the character cluster "/)/" is divided into two tokens "/)" and "/". We have avoided attaching any special significance to the characters of an infix operator to allow infix tokens such as "//" and "..". Note that postfix characters may only be the last character of a token.

Bifix tokens are used as both prefix and postfix operators. An example of this is "↑" in Pascal, which is a prefix operator in type declarations and a postfix operator in expressions. In general, the first character of a bifix token must be a prefix character, and the last must be postfix. If a bifix token contains only one character, it is both a prefix and postfix character. We might call such a character a bifix character. Bifix characters may not be part of other tokens, as they must be both the first and last character.

We may see now why special-character operands must be bifix tokens. They must begin with a prefix character for separation from a preceding infix operator, and must end with a postfix character for separation from a following operator.

Evaluation of the Operator Scanner

While the use of character classifications to break apart character clusters might seem overly simple, it is a very effective method. Our goal is the development of a scanner having definable tokens. To evaluate the method, we can use the operators found in existing languages. This does not imply that our goal is to make a system for processing all existing languages; these languages serve only as a source of data, providing operator tokens which have been useful in the past.

Pascal, PL/I, and Algol 68 can all be handled by the scanning technique. Some of the character classifications might not be obvious. In PL/I, for example, "(", ",", and ")" must all be bifix characters. This allows proper handling of the strings "DCL A (*,*)" and "A**B". A similar classification is needed for Algol 68, where an array with a single dimension might have the string "()" as part of its type name. In Pascal, with "(." and ".)" used in place of square brackets "[" and "]", we would need a special provision in the grammar for the empty set: "(. .)" and "(..)" are both valid representations, although the latter is only one token. "." could be classified as a bifix character, but ".." would become two tokens while "(..)" would become four.

Simula 67 has the problem described earlier regarding the token ":-", but other special symbols such as "==" and "=/=" are handled correctly. The grammar for Simula could be modified to read ":-" as two tokens, or to allow ":-" as a single token in array declarations. Note that if ":-" is two tokens, the compiler would also accept ": -".

Rather than looking at additional languages by themselves, let's consider several at one time. The special character tokens from the following languages were accumulated: Pascal [13], Algol 68 [21], Algol 60 [16], Cobol [12], Simula 67 [5], Simscript 2.5 [2], IMP (Edinburgh) [1], Jovial [20], Snobol [8], Sue [3], BCPL [17], C [18], Mary [4], and Euclid [15]. We shall select a specific classification of the characters, and use it with operators from all of these languages. The character classification is shown below:

Prefix characters: + - ([{ ↑ # ! @ ? , ;] }

Postfix characters:)] } ↑ ! @ # ? , ; [{

The table below shows those tokens from the selected languages which can be recognized as such with the character

classification above:

```
Prefix
     +  -  ↑  (  (*  (/  ($  (.  (:  [  {  #  !  @  ?
Infix
     +  -  *  /  **  //  |  &  ||  &&  ¬  #  @  .  ..  ...  ?  %
     =  <  >  <=  >=  <>  ><  ¬=  ¬<  ¬>  =>  :=:  :/=:  ==  =/=
     :=  +:=  -:=  *:=  /:=  %:=  %*:=  =:  ->  +=:  =*  =%  =<<
     =>>  =&  =|  =:*  =:**  =:/  =://  =:<<  =:>>  =:<*  =:*>
     <<  >>  +>  <*  *>  |:  <<=
Postfix
     )  *)  /)  $)  .)  :)  ↑  ]  }  ?  !  @
Bifix
     ,  ;  #  [  ]  (..)  ()
```

The symbols labelled "Bifix" require some explanation. In several of the languages, it is possible to omit items from a list. This leads to several adjacent "," or ";" symbols. These tokens were classified as "bifix" so that they don't combine with each other or adjacent brackets. For empty lists, we may make the brackets into bifix symbols, so that "[ ]" becomes two tokens. In the case of "(" and ")", we cannot use bifix symbols, since we desire multi-character brackets (like "(/"). We may still handle the null lists by accepting "()" as an alternative to "(" ")". These problems arise because the languages allow lists with omitted elements, which do not follow the rules of operators stated earlier.

The "#" is bifix for a different reason. In the IBM 370 extensions to BCPL [17], operators such as "#+" and "#-" are used as floating point operators. The "#" symbol transforms integer operators into floating operators. Rather than regard "#+" and "#*" as new operators, the "#" was considered to be a functor which transformed the operator. The functor must be a postfix token due to the operator to its right. It must also be a prefix token when "#-" is used as a prefix operator.

The table below shows those tokens which are not handled properly:

```
Prefix
     <  $(  .  $  %  *  /  &  |  ++  --
Infix
     :-  ++  <+  <-  !!  =+  =-  !=  =↑  =:+  =:-  $!
     =:<+  =:+>  ,.
Postfix
     >  %  ++  --  -  =>  <<  >>  **  <+
Bifix
     *  (  )
```

Let's consider why these symbols could not be tokens. We have already discussed the exclusion of ":-", "++", and "--",

operators which introduce lexical ambiguity in a general processor. A number of the prefix exclusions were introduced by Snobol, which uses blanks to distinguish between infix and prefix operators. Many of these prefix operators are "spares". The language IMP introduced the tokens "<-" and "!!". The first of these is clearly ambiguous in our scheme, while "!!" was excluded because the postfix use of "!" seemed more important.

In addition to the prefix and postfix "++" and "--", C provided the operators "=+", "=-", "!=", and "=↑". The first two are ambiguous in the language C, leading to different interpretations for "A =- B" and "A = - B". "!=" and "=↑" are excluded by other desired uses for "!" and "↑".

Mary provided the greatest challenge for the scanner, because Mary uses postfix operators where other languages use prefix operators. For a specialized Mary scanner, we would redefine "+" and "-" as postfix instead of prefix characters. This would enable us to handle the postfix operators "-", "*+", and "<+", as well as the infix operators "=:+", "=:-", and "=:<+". Some Mary operators could not be handled. In fact, Mary programmers can define any sequence of characters from the set "+-*/%&<=>!.:" as an operator.

Several of the "spare" operators of Algol 68 were omitted from this table. They could not be handled by the scanner only because we chose not to make "&" and "%" prefix characters, and because "↑" is a bifix character. Some operators of Algol 68 disappeared because they are outside of the character set being used for this classification.

APL [9] was originally in our language sample, but was discarded. With all of the symbols which it uses, it does not compose operators from multiple characters. Note that operators such as "+.*" have several characters only because "." is being used as a functor, combining the infix operators "+" and "*". The local version of APL [19] leads to additional troubles when the keypunch character set is used. The extra symbols are encoded as "$" followed by one or two characters. At times these symbols are letters. In some cases the "$" may be regarded as a functor, but in others the scanner cannot handle this type of token.

Other Tokens

So far we have considered only the special character tokens. All of our work to ensure lexical non-ambiguity will be lost if the other tokens of the language are ignored. The additional classes of tokens which shall be considered here are identifiers, integers, reals, and strings. In addition, the scanner must be able to recognize comments. Definitions shall be given for each of these classes. These definitions should not be considered as universal, but as a formulation chosen for

the sake of analysis. Personal preference might suggest a number of modifications to these definitions, most of which could be adapted for this scanning technique.

Tokens are defined in terms of the characters which form them. The characters being used for processing are divided into five disjoint classes: letters, digits, break characters, quotes, and special characters.

For convenience, we shall call special character tokens operators. We may extend our previous definition by allowing break characters as any character of an operator other than the first. Certain break characters might be distinguished as postfix characters if desired.

Integers are defined as sequences of digits. For our purposes, there are no limitations on the number of digits in an integer. Integers are unsigned, as a sign cannot be regarded as a digit. (If a sign was a digit, it could appear anywhere in an integer.) The parser could associate a prefix operator and an integer to form a signed value.

Identifiers always begin with a letter, but otherwise may be any sequence of letters, digits, and break characters. Identifiers may appear next to any operator, because a letter may not appear in an in an operator, and a special character may not appear in an identifier. If break characters were allowed to begin an operator, we would not be able to place such an operator immediately after an identifier. Identifiers must be separated from other identifiers and from integers (and reals) by spaces. Identifiers might be used for keywords or operators by the parser; this is irrelevant to the scanner.

Real numbers pose a number of problems for the scanner. There are two common formulations of real numbers. The first requires that there be one or more digits on each side of the decimal point, while the second allows a decimal point to start or end a number. If we use the first formulation, the decimal point character may be included in the special character set. If the second formulation is used, the decimal must be excluded from all of the character classes. If the decimal point could start an operator, then "12." could be the number "12" followed by a "." operator. "." can't be a break character, unless such characters are excluded from the construction of operators, or "(.12" could be divided into "(." and "12" or into "(" and ".12".

The second problem with reals is the exponent portion. This consists of a "power of ten" symbol, an optional sign, and an integer. The simplest approach regards the "power of ten" symbol as being an infix operator of the highest priority, and the sign as being a prefix operator in front of the exponent. The resulting definition of real numbers would differ from the traditional definition, since blanks could appear before the

exponent portion. Unless the definition of the "power of ten" symbol as an operator is valid only after an integer or decimal number, it would be available as an operator between an arbitrary real or integer and an integer. At least with the Unlanguage parser, it would be possible to eliminate this type of usage.

In certain situations the traditional form of real numbers might be desired. The complete real constant would then be a single token. The safest way of achieving this would be to reserve a unique "power of ten" symbol which appears in none of the character classes, or occurs only as a break character. The common practice of allowing the letter "E" as the "power of ten" symbol can be dangerous if identifiers are allowed to follow numbers with no intervening spaces. If "E1" is an identifier, then "1.2E1" could be either a real number alone, or the real "1.2" followed by the identifier "E1". Our rule, then, is that whenever the "power of ten" symbol belongs in some character class, a token which begins with a character of that class must be separated from a real number or integer by one or more spaces.

Strings and comments are tokens which are enclosed by quotes. In order to make some practical decisions about the formats of these tokens, we shall make a few arbitrary restrictions. Strings shall be enclosed by quotes which are single characters, and the opening and closing quote characters are identical. Comments, on the other hand, may have quotes which are either single or multiple characters, and the closing quote may differ from the opening quote. In addition, a variant of the comment will permit the end of the input line to serve as the closing quote. Comments may run from line to line, while strings must appear on a single line only. The value of the string token is available, while comments are eliminated by the scanner. There may be a number of different types of strings, where the quote symbol selects between the different types.

Strings are simpler to process than comments. The quote symbols used to delimit strings cannot appear in any other class of characters, so strings may appear next to other tokens. For simplicity, we shall assume that a doubled quote character represents a single occurrence of that character within the string. Escape characters could be specified to allow control functions, but different escape characters may be needed for each string type, which would complicate the description of the scanner. Note that the definition of a postfix operator "B" would allow the conversion of one string type to another. For example, '01'B could specify that '01' is a character string, while the B could convert this to a bit string.

Comment brackets may be arbitrary clusters of characters, with other character clusters serving as a comment close. We could, for example, define "/*" and "*/" to be a pair of comment brackets, or we could use "{" and "}". Once again, it is safest

if the first character of the left bracket cluster is used for no other purpose in the language, as in the Pascal use of "{". When a comment bracket like "(*" is used, the first character of the bracket is a prefix character, so the symbol is safe as a comment start. If "/*" is permitted as a comment bracket but "/" is an infix character, then the user might attempt to define clusters such as "//*" but would find that a comment had been started. The scanner would have a more difficult time finding this comment bracket than if it only needed to check at the beginning of a token. This suggests that comment brackets must be separated from other tokens by a space if they might otherwise be mistaken as part of that token. But since special character tokens do not need space separation in other constructs of the language, the user might expect that no such a space is needed. We could be a bit more conservative and require that an open bracket of a comment be a bifix token. If "/*" was a bifix token, the user could not define a token such as "/*/" which starts out with the comment open bracket. Any character or cluster may be used as a closing bracket. But note that if the cluster is not a bifix token, a similar operator (such as "*/*") might appear in the comment, causing accidental termination of the comment string. Likewise, if the closing quote is used elsewhere in the language (as ";" is in ALGOL), a portion of code may not be placed conveniently within comment brackets.

Implementation

One of our stated goals was efficiency, so we should consider the implementation of our scanner. Except for some uncertainty regarding comments and real numbers, the first character of the token dictates the token type. The end of the token is either a blank, a postfix character, or the first character of the next token. Because the token separation is based solely on properties of the characters, we may use a scanning process to separate tokens instead of a finite-state machine. The scanning process may be carried out by a table-driven scan which resembles the IBM 370 TRT instruction. This routine could be micro-coded on appropriate hardware. It requires nine instructions on the IBM 370 (including a TRT), and occupies about 20 bytes on an INTEL 8080.

Comparison

The Unlanguage scanner has a number of advantages over each of the three traditional scanning methods: the finite-state scanner, the "Snobol technique", and the "Mary technique". It has a number of disadvantages as well.

The Unlanguage scanner has one basic advantage over each of the other techniques: the finite-state scanner cannot be extended as easily, the "Snobol technique" is less convenient

for the programmer, and the "Mary technique" may lead the programmer into ambiguous constructs. A finite-state machine could, of course, be used to implement any of these techniques, including the unlanguage scheme. We are comparing the unlanguage scanner with the traditional method of encoding specific tokens within the machine.

Probably the greatest advantage of the Unlanguage scanner is its protection against lexically ambiguous constructs. It makes it possible for a language designer to permit operator name extension but still provide protection against the construction of lexically ambiguous languages.

The Unlanguage scanner is also quite fast. Its principal scan procedure can be micro-coded. It does not require a large amount of space. The principal tables characterize the characters based on their occurrence as the first character of a token and their use after the first character. These tables require approximately two bytes per character. An additional table gives matching close brackets for comment open brackets.

The major disadvantage of the Unlanguage scanner is its restrictions on operator use. The characters must be classified for the entire language, and cannot change their qualities in different places within the language. This restriction does not seem to be serious, as can be seen by the operator analysis of existing languages. It would be possible to switch scanning tables during a compilation if the proper controls are provided at a higher level.

Bibliography

[1] Barritt, M. et al., Edinburgh IMP Language Manual, Edinburgh Regional Computing Centre, July 1970.

[2] CACI, Simscript II.5 Reference Handbook, CACI, Los Angeles (1973).

[3] Clark, B. L., The SUE System Language User's Guide, Computer Systems Research Group, University of Toronto (Revised July 1977 by A. Ballard, University of British Columbia.)

[4] Conradi, R. and P. Holager, MARY Textbook, RUNIT report STF14A74034, University of Trondheim, Norway (1974).

[5] Dahl, O. et al., Common Base Language, Norwegian Computing Center Pub #S-22 (1970).

[6] Fraley, R., "Unlanguage Grammars and their Uses", Department of Computer Science Technical Report 77-6, University of British Columbia, (in preparation).

[7] Gries, D., Compiler Construction for Digital Computers, John Wiley and Sons, New York (1971).

[8] Griswold, R. et al., The SNOBOL 4 Programming Language, Prentice Hall, New Jersey (1971).

[9] IBM, APL/360 User's Manual, IBM, GH20-0683-1, (1970).

[10] IBM, FORTRAN IV Language, IBM Corporation, GC28-6515-8, (1971).

[11] IBM, PL/I(F) Language Reference Manual, IBM GC28-8201-4 (1972).

[12] IBM, DOS Full American National Standard COBOL, IBM Corporation, GC28-6394-4, (1973).

[13] Jensen, K. and N. Wirth, Pascal User Manual and Report, Springer-Verlag New York (1976).

[14] Johnson, W. et al., "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques", CACM, v.11, n.12, Dec. 1968.

[15] Lampson, B. et.al., "Report on the Programming Language Euclid", Sigplan Notices, v.12, n.2,

Feb.,1977.

[16] Naur, P. ed., "Revised Report on the Algorithmic Language Algol 60", CACM, v.6, n.1, Jan. 1973.

[17] Richards, M., The BCPL Programming Manual, Computer Laboratory, University of Cambridge, April 1973; as adapted by J.E.L. Peck for MTS, University of British Columbia, July 1975.

[18] Ritchie, D., C Reference Manual, Bell Labs, Murray Hill, N. J. (1974).

[19] Twiver, D., APL - A Programming Language, University of British Columbia Computing Centre, Manual UBC APL, Sept. 1971.

[20] Univac, JOVIAL Programmer Reference, Sperry Rand Corporation, UP-7698 (1973).

[21] van Wijngaarden, A. et al., Revised Report on the Algorithmic Language Algol 68, Springer-Verlag, New York (1976).