REFERENCES

1. GRAY, J. C. Compound data structure for computer aided design; a survey. Proc. ACM 22nd. Nat. Conf. 1967, Thompson Book Co., Washington, D. C., pp. 355–365.
2. HANSEN, W. J. The impact of storage management on the implementation of plex processing languages. Tech. Rep. No. 113, Computer Science Dep., Stanford U., Stanford, Calif., 1969.
3. KNUTH, D. E. *The Art of Computer Programming, Vol. 1.* Addison-Wesley, Menlo Park, Calif., 1968.
4. LANG, C. A., AND GRAY, J. C. ASP—A ring implemented associative structure package. *Comm. ACM 11*, 8 (Aug. 1968), 550–555.
5. MCCARTHY, J., ET AL. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.
6. MINSKY, M. L. A LISP garbage collector using serial secondary storage. MIT Artificial Intelligence Memo. No. 58, MIT, Cambridge, Mass., Oct. 1963.
7. Ross, D. T. A generalized technique for symbol manipulation and numerical calculation. *Comm. ACM 4*, 3 (Mar. 1961), 147–150.
8. ——. The AED free storage package. *Comm. ACM 10*, 8 (Aug. 1967), 481–492.
9. REYNOLDS, J. C. Cogent programming manual. Argonne Nat. Lab. Rep. No. ANL-7022, Argonne, Illinois, Mar. 1965.
10. SCHORR, H., AND WAITE, W. M. An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM 10*, 8 (Aug. 1967), 501–506.
11. STYGAR, P. LISP 2 garbage collector specifications. TM-3417/500/00, System Development Corp., Santa Monica, Calif., Apr. 1967.
12. WISEMAN, N. E. A simple list processing package for the PDP-7. In *DECUS Second European Seminar*, Aachen, Germany, Oct. 1966, pp. 37–42.

A Base for a Mobile Programming System

RICHARD J. ORGASS
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

AND

WILLIAM M. WAITE
University of Colorado, Boulder, Colorado

An algorithm for a macro processor which has been used as the base of an implementation, by bootstrapping, of processors for programming languages is described. This algorithm can be easily implemented on contemporary computing machines. Experience with programming languages whose implementation is based on this algorithm indicates that such a language can be transferred to a new machine in less than one man-week without using the old machine.

KEY WORDS AND PHRASES: bootstrapping, macro processing, machine independence, programming languages, implementation techniques
CR CATEGORIES: 4.12, 4.22

1. Introduction

The development of many special purpose programming languages has increased the overhead associated with changing computing machines and with transferring a programming language from one computer center to another. A number of writers have proposed that these languages should be implemented by bootstrapping. (Since this work is well known, it is not summarized in this introduction.)

The description is given of a bootstrapping procedure which does not require a running processor on another machine for its implementation. A compiler is described which can be trivially implemented "by hand" on contemporary computing machines. This simple compiler is then used to translate a more elaborate one, and so forth, until the desired level of complexity is reached.

This paper is a complete description of the first stage of such a bootstrapping procedure. It is organized as follows: in Section 2, the processing performed by the simple compiler (SIMCMP); in Section 3, some examples of its use; in Section 4, the environment which must be coded for a particular computing machine; and in Section 5, the SIMCMP algorithm.

2. Specifications for SIMCMP

The algorithm described in this paper was constructed to provide a compiler of minimum length and complexity which is adequate to serve as the base for the implementation, by bootstrapping, of programming systems. SIMCMP is essentially a very simple macro processor. Source language statements are defined in terms of their object code translations. These definitions are stored in a table. A source program is translated one line at a time. The input line is compared with all of the entries in the table. When a match occurs, the object code translation of the input line, with appropriate parameter substitutions, is punched. SIMCMP can be used to translate any source language which can be defined by means of simple substitution macros with single character parameters.

Several terms as they are used in this paper are illustrated here. In a conventional macro processor, a *macro definition* is of the form:

MACRO NAME (P_1, $\cdots$, P_n)
Code body
END

The strings 'MACRO' and 'END' serve to delimit the macro definition. 'NAME' stands for the *name of the macro*, and 'P_1', $\cdots$, 'P_n' stands for the *formal parameters* of the macro. This macro is *called* by a line of the form 'NAME (A_1, $\cdots$, A_m)' where m is less than or equal to n. When this *call* is encountered in the program text, the code body of the macro NAME is *evaluated* by sub-

stituting the values of 'A_1', $\cdots$, 'A_m' for occurrences of 'P_1', $\cdots$, 'P_n'. If m is less than n, then values for *actual parameters* 'A_{m+1}', $\cdots$, 'A_n' are generated by the macro processor in some regular way.

SIMCMP differs from a conventional macro processor in that the line which introduces a macro definition may be an arbitrary string of characters, with parameters indicated by a special symbol called a *parameter flag*. This line is referred to as the *template*. The effect of using a template is to allow the name part of a traditional macro to be replaced by a "distributed name" which consists of all the characters in the line except the parameter flags. Except for the parameter flags, each character of the template must match the corresponding character of the input line exactly. A parameter flag, however, may match any single character. That is, the template line may be thought of as a mask consisting of literal characters interspersed with "holes" which correspond to arbitrary characters in the input string being matched. The character which is matched by a particular parameter flag becomes the value of the formal parameter which is denoted by this parameter flag.

The code body of a SIMCMP macro consists of lines in the target language with references to parameters substituted for some elements of the lines. That is, each line consists of a series of strings of characters and references to parameters. A reference to a parameter is signaled by a special character called a *machine language* (MCT) *parameter flag*. This flag is followed by two digits, the first of which specifies the number of the formal parameter in the template, counting from the left. The second digit of the parameter call defines the type of conversion to be used in the particular substitution instance of the formal parameter. A maximum of nine formal parameters, numbered 1 to 9, may be specified in a single template. The two conversion types which are available are described below.

The SIMCMP translator has two main phases: macro definition and macro expansion. During the macro definition phase, user-defined macros are read and stored in an array. The first input line for SIMCMP is called the *flag line* and contains five control characters. The first is the *source language end-of-line flag*, a character which marks the end of characters to be translated in an input line. The second character is the *source language parameter flag* which is used in templates to indicate the position of formal parameters. The third and fourth characters of the flag line, respectively, are the MCT *end-of-line* and the MCT *parameter flags*. The first marks the end of useful information in a code body line; the second indicates that a converted actual parameter should be inserted at this point in the machine code output. The fifth character on the flag line must be the character '0'. Following the flag line is a series of macro definitions. Each macro definition is terminated by a line whose first character is the MCT end-of-line flag. After this line, a new template must appear. If a particular macro is the last one to be defined, then the first *two* characters of its terminating line are machine code end-of-line flags.

After all of the macro definitions have been read, SIMCMP enters the expansion phase. A single source statement is read into the array. Reading is terminated when a source end-of-line flag occurs in the input stream. This statement is translated by successively matching it against each of the defined templates. Before this comparison is done, a check is made to ensure that the source line is not void (a void line terminates the program to be translated). During the matching process, a character which is matched against a source language parameter flag in the template is placed in the corresponding parameter storage word. If all the characters in the input line are successfully matched, the input line is accepted as a call on the macro whose template is currently being scanned. If SIMCMP fails to find a match for some character of the input line, an attempt is made to match this input line to the next template. If all templates have been compared with an input line and there is no match, then the input line is assumed to be in the target language and it is punched out without translation. After processing an input line, the next input line is processed, and so forth, until a viod line is encountered.

Since the templates are scanned in order, an input line which matches several templates will be treated as an instance of the first template encountered. Therefore, it is the responsibility of the user to ensure that the ordering of the macro definitions will not produce strange results.

When an input line has been matched to a particular template, the lines of its associated code body are punched out. Characters other than the formal parameters are punched exactly as they appear in the code body and call on formal parameters are replaced by the values specified by the conversion digit and the actual parameter value. Two conversion types are available: type 0 and type 1. Type 0 conversion is a direct copy of the actual parameter into the output string. For each computing machine, the user of SIMCMP must define a one-to-one mapping, M, from the character set of his computing machine onto a set of positive integers. The only restriction on M is that the characters 0 to 9 must map onto successive integers. Type 1 conversion produces as output the numeral which denotes the image of the actual parameter under the mapping M.

For some target languages, it is necessary to SIMCMP to be able to generate arbitrary, unique symbols. This capability is provided by means of parameter zero. Up to ten unique symbols may be generated during the expansion of a single macro by referring to parameter zero with conversion type 0 to 9. The conversion type, in this case, indicates which of the created symbols is being referred to (for example, '00' refers to the first created symbol, '01' refers to the second created symbol, etc.). These symbols are three-digit decimal numerals; the first one denotes the integer 100. These numerals are supplied by a "location counter" which is maintained by SIMCMP. This location counter is initially set to the value 100; after processing a macro which contains references to parameter

0, its value is incremented by one more than the highest conversion digit used.

The structure of the translation performed by SIMCMP imposes restrictions on both the source language and the target language. Restrictions on the source language syntax are not important because the only purpose of SIMCMP is to compile the next compiler in the bootstrap sequence. Restrictions on the target language are more serious; so we have attempted to generalize the program without compromising our design objectives.

3. Examples

The compiler for the list processing language WISP [1] is written in a subset of itself which can be compiled by SIMCMP. SIMCMP has been used to implement the WISP language for several computing machines. The programming effort required to implement WISP in this manner is about one man-week. The macro processor LIMP [2] is written in the programming language WISP and, consequently, is available once WISP has been implemented.

WISP may be described approximately by saying that it is a simple version of the "program feature" of LISP [3]. The examples given here are taken from the subset of WISP which is compiled by SIMCMP. In order to provide examples which do not refer to a particular computing machine, the FORTRAN II programming language is used as the target language. In actual practice, the target language would be the assembly code of a particular machine.

Figure 1 contains examples of two SIMCMP macros and illustrations of the output produced by SIMCMP for particular source language strings as inputs. In this figure, the source language and MCT end-of-line flags are "·". The source language parameter flag is "*" and the MCT parameter is "'".

An example of the use of parameter 0 is given in Figure 2. The flags in this figure are the same as those in Figure 1. Since the target language is FORTRAN II, arbitrary labels must be generated because three labels must be specified for the IF statement. When the condition is not satisfied, control is to pass to the next statement, which must be assigned a label. SIMCMP produces this label automatically by means of parameter 0.

4. The Environment for the Simple Compiler

The environment for SIMCMP is defined to be all programs which are required to use SIMCMP on a particular computing machine. That is, the environment includes input and output programs, the assembler for this computing machine, and a driver program. (The SIMCMP algorithm is given as a procedure.) The input program is used to read from a source created by the user. This source is to be organized into lines. For remote terminal or paper tape oriented machines, a line is delimited by carriage returns. For card oriented machines, each card is a line and is considered to have a carriage return following the 80th character.

The output program is used to write text which is to be processed by the assembler; the output text is also organized into lines.

The input program (IREAD) is an integer procedure with one parameter. The value returned by this procedure depends on the value of the parameter and the next character in the source. The parameter of IREAD may have the value 0 or 1. If IREAD is called with a parameter whose value is 1, then the value returned by

```
* = CAR.*.
      I = CDR('21).
      CDR('11) = CAR(I).
.X
```

(a) Example of SIMCMP macro definition.

```
A = CAR B.
```

(b) A string which matches the template of (a).

```
I = CDR(38)
CDR(36) = CAR(I)
```

(c) A possible output from macro (a) when string (b) is the input

```
*** * = *** *.
      I = CDR('41).
      J = CDR('81).
      '10'20'30(I) = '50'60'70(J).
.X
```

(d) Another example of a SIMCMP macro definition

```
CAR A = CDR B.
```

(e) A string which matches the template of (d).

```
I = CDR(36)
J = CDR(38)
CAR(I) = CDR(J)
```

(f) Output produced by SIMCMP when string (d) is the input.

FIG. 1. Examples of SIMCMP macros

```
TO ** IF CAR *¶= CDR *.
      I = CDR('31).
      J = CDR('41).
      (IF (CAR(I) - CDR(J))'00, '10'20, '00.
'00   CONTINUE.
.X
```

(a) A SIMCMP macro definition which uses parameter zero.

```
TO 13 IF CAR A = CDR B.
```

(b) A string which matches the template of (a).

```
      I = CDR(36)
      J = CDR(38)
      IF (CAR(I) - CDR(J))100, 13, 100
100   CONTINUE
```

(c) Output produced by SIMCMP if string (b) is the first input string which matches a template whose definition uses parameter zero.

```
TO 14 IF CAR B = CDR A.
```

(d) A string which matches the template of (a).

```
      I = CDR(38)
      J = CDR(36)
      IF (CAR(I) - CDR(J))101, 14, 101
101   CONTINUE
```

(e) Output produced by SIMCMP if string (d) is the second input string which matches a template whose definition uses parameter zero.

FIG. 2. Example of the use of parameter zero

the procedure is the integer which corresponds to the next character in the source under the mapping M. Note that it is assumed that the source is organized so that the first character of a line immediately follows the carriage return of the line which precedes it. The carriage return itself corresponds to the integer -1.

If IREAD is called with a parameter whose value is 0, then the value returned by the procedure is undefined.

```
      SUBROUTINE SIMCMP(LIST,KMAX)
      DIMENSION LIST(KMAX)
C     READ CONTROL CHARACTERS, IN THE ORDER -
C                   SOURCE EOL, SOURCE PARAM, MCT EOL, MCT PARAM, ZERO.
      DO 1 I=1,5
    1 LIST(I)=IREAD(1)
      LIST(6)=100
      K=17
C     READ MACRO DEFINITIONS
    2 I=IREAD(0)
      L=K+1
      LIST(L)=-1
      I=L
    3 IF (I .GE. KMAX) STOP
      I=I+1
      LIST(I)=IREAD(1)
      IF (LIST(I) .NE. LIST(1)) GO TO 3
    6 J=IREAD(0)
      J=I+1
    7 I=I+1
      IF (I .GE. KMAX) STOP
      LIST(I)=IREAD(1)
      IF (LIST(I) .NE. LIST(4)) GO TO 12
      LIST(I)=LIST(5)-IREAD(1)-7
      I=I+1
      LIST(I)=IREAD(1)-LIST(5)
      IF (LIST(I-1) .NE. (-7)) GO TO 7
      IF (LIST(L) .LT. LIST(I)) LIST(L)=LIST(I)
      GO TO 7
   12 IF (LIST(I) .NE. LIST(3)) GO TO 7
      LIST(I)=-1
      IF (I .NE. J) GO TO 6
      LIST(K)=I
      K=I
      IF (IREAD(1) .NE. LIST(3)) GO TO 2
C     READ A SOURCE STATEMENT
   20 I=IREAD(0)
   21 DO 22 I=K,KMAX
      LIST(I)=IREAD(1)
      IF (LIST(I) .EQ. LIST(1)) GO TO 30
      IF (LIST(I) .EQ. (-1)) GO TO 52
   22 CONTINUE
      STOP
C     TRANSLATE ONE STATEMENT
   30 IF (I .EQ. K) RETURN
      M=17
   31 L=8
      N=M+1
      DO 34 J=K,I
      N=N+1
      IF (LIST(N) .EQ. LIST(2)) GO TO 33
      IF (LIST(N) .EQ. LIST(J)) GO TO 34
   32 M=LIST(M)
      IF(M-K)31,50,50
   33 IF (I .EQ. J) GO TO 32
      LIST(L)=LIST(J)
      L=L+1
   34 CONTINUE
      GO TO 41
C     PUNCH MACHINE CODE TRANSLATION
   40 CALL IPNCH(LIST(N))
   41 N=N+1
      IF (LIST(M) .EQ. N) GO TO 47
      IF (LIST(N) .GE. (-1)) GO TO 40
      L=-LIST(N)
      N=N+1
      IF (L .EQ. 7) GO TO 42
      IF (LIST(N) .NE. 0) GO TO 43
      CALL IPNCH(LIST(L))
      GO TO 41
   47 LIST(6)=LIST(M+1)+LIST(6)+1
      GO TO 20
C     CONVERT A PARAMETER TO AN INTEGER
   42 LIST(7)=LIST(N)+LIST(6)
   43 I=LIST(L)
      DO 44 J=K,KMAX
      L=I/10
      LIST(J)=I-(L*10)
      IF (L .EQ. 0) GO TO 45
   44 I=L
      STOP
   45 CALL IPNCH(LIST(J)+LIST(5))
      J=J-1
      IF (J .GE. K) GO TO 45
      GO TO 41
C     PUNCH OUT AN UNRECOGNIZED LINE AFTER GETTING IT ALL IN
   50 J=I+1
      DO 51 I=J,KMAX
      LIST(I)=IREAD(1)
      IF (LIST(I) .EQ. (-1)) GO TO 52
   51 CONTINUE
      STOP
   52 DO 53 J=K,I
   53 CALL IPNCH(LIST(J))
      GO TO 21
      END
```

FIG. 3. The SIMCMP algorithm

However, this call causes the input stream to be moved forward until a carriage return character has been passed. That is, after such a call, the next call to IREAD with a parameter whose value is 1, returns as its value the integer which corresponds to the first character of the next input line. Successive calls IREAD(0) may be used to skip input lines.

The output program (IPNCH) is a procedure with one argument which is used to convert integers to the characters to which they correspond. The statement CALL IPNCH(I) causes the character which corresponds to I to be placed in the output character stream. Since the integer -1 corresponds to the carriage return, CALL IPNCH(-1) terminates the current output line. That is, for a card oriented machine it causes the termination of the current output card; for a remote terminal or paper tape oriented machine, it causes the carriage return character to be placed in the output stream.

By contemporary standards, the assembler required for the SIMCMP environment is quite simple. This assembler must be capable of assigning values to symbolic addresses which are strings built up out of numerals or numerals and other characters. The symbols generated by SIMCMP are always strings of numerals. However, the translation rules may be written so that a string of numerals is preceded or followed by another string of characters. This assembler must have a facility for reserving storage.

The driver program defines the storage used by the procedure SIMCMP and calls this procedure. It may also be used to take care of bookkeeping functions. During translation, SIMCMP requires a single integer array and six temporary storage locations. The array must be large enough to contain all of the translation rules (stored one character per element) and a single input line. In addition, fifteen elements of the array plus two elements per translation rule are used for control information.

5. The SIMCMP Algorithm

The SIMCMP algorithm is shown in Figure 3. This algorithm is stated as a FORTRAN IV subroutine. It has been used in this form with an IBM 7044, an IBM 360/50, and a CDC 6400.

The two parameters are the array described in Section 4 (LIST) and the number of elements in this array (KMAX). The comments in the program explain its operation, although this information is not required for the use of the program.

RECEIVED DECEMBER 1967; REVISED MAY 1969

REFERENCES

1. ORGASS, R. J., SCHORR, H., WAITE, W. M., AND WILKES, M. V. WISP—A self-compiling list processing language. Tech. Rep. No. 36, Basser Computing Dep., U. of Sydney, Australia, Oct. 1965. (Also reprinted by the Dep. of Electrical Engineering, U. of Colorado, Boulder, Col., and by the Columbia U. Computer Center, New York.)
2. WAITE, W. M. A language independent macro processor. *Comm. ACM 10*, 7 (July 1967), 433–440.
3. MCCARTHY, J. ET AL. LISP 1.5 programmer's manual. MIT Computation Center, Cambridge, Mass., 1962.