Serdica
Bulgariacae mathematicae publicationes

Сердика
Българско математическо списание

EXTENSIONS OF A FUNCTIONAL LIST PROCESSOR

GEORGI I. POPOV

Extensions of a functional list processor are proposed, discussed, and specified on the WISP language. The processor itself is the HELP processor and its principles, workings, and machine-independent implementation are described in W e i t e (1973). The extensions are supposed to widen the application field of the processor, to facilitate the programming of sophisticated problems, and to allow for the execution of large programs in a memory limited in size.

1. Introduction. A prototype of the HELP processor is LISP, but more natural external notation of the procedures is adopted and some of the features are restricted. Brief notes regarding some of the HELP characteristics would be useful to provide a basis for the explanation of the extensions proposed.

In HELP a notational distinction is made between constants and variables. A HELP constant is an atom defined as any string of letters, digits and asterisks, the first being an asterisk. A HELP variable is defined similarly except that the first character must not be an asterisk. The variables are used as names of functions or bound variables in function definitions.

A HELP program is a set of symbolic expressions to be evaluated. The syntax of the HELP expressions can be found in [1].

The functions form the heart of the HELP processor. There are five built-in functions:

the selectors CAR(e) and CDR(e),

the constructor CONS (e_1, e_2),

the predicates ATOM(e) and NULL(e).

All other functions used must be defined by the user. They permit the problem to be divided into components while programming is going on, but the evaluation of an expression is an indivisible process. Any problem should be formulated in one expression only and all input data must be supplied with that expression. Sometimes this is not so convenient. Moreover, if the available memory is not sufficient for the evaluation of an expression corresponding to a complex problem the problem has to be divided into several parts. These parts are to be programmed with different expressions. Since a data transfer between expressions is not possible, several completely independent runs of the HELP processor are necessary, because the HELP processor has no explicit storage for temporary allocation of intermediate results.

In the proposed modified HELP version such an explicit storage is provided. A result produced by one expression could be used by others. Thus, the user's HELP program becomes an ordered sequence of expressions.

2. Global variables and assignment statements. For the purposes of data transfer between expressions, a special global type of a variable with corres-

SERDICA Bulgaricae mathematicae publicationes. Vol. 3, 1977, p. 94—105.

ponding notational distinction is introduced. A global variable is represented by any string of letters, digits and asterisks, beginning with &. Examples of global variables are :

& GV1
& 155
& THIS * IS * A * GLOBAL * VARIABLE

The value of any global variable can be an atom, a list, or remain undefined in some cases. A global variable dictionary is used to associate the names of the global variables with their values. This dictionary provides the explicit storage mentioned above.

The global variables could be used as primaries in the HELP expressions. An assignment statement is added to the HELP syntax to assign values to the global va iable and the right part could be any type of expression. The value computed by one expression and assigned to a global variable could be used in another expression as a value of a primary. Hence, the syntax of HELP is extended with a global variable and an assignment statement. The modified HELP becomes more like a conventional programming language.

The global variable dictionary is addressed by the base register G. This register must be reset at the beginning of a program execution.

```
        A=NIL, F=NIL, G=NIL.                          X HELP∅∅∅2
```

The character X before the sequentional number of a line (for example, X HELP∅∅∅2) means that the line of the source WISP text of he processor HELP with that sequentional number is changed. When an insertion is made, a digit instead of X is put to indicate the index number of the inserted line.

The original HELP subroutine FINPUT is modified to pick up the global variables.

```
        TO GVAR IF (CHAR)='&.                          1 HFLP∅478
GVAR,   USE NEXTCH, USE READST.                        2 HELP∅515
        D='G, M='X, USE LOOKUP.                        3 HELP∅515
        TO BAKG IF CDR Y NE NIL.                       4 HELP∅515
        Z=NEW ELEMENT, CDR Y=Z.                        5 HELP∅515
        CAR Z=NIL, CDR Z=S.                            6 HELP∅515
BAKG,   Z=2∅, EXIT FINPUT.                             7 HELP∅515
```

The syntax type of a global variable is assumed to be 2∅. After recognition of a global variable the control is transferred to the label ASST where a check up for an assignment statement is performed.

```
        TO ASST IF Z=2∅.                               1 HELP∅∅5∅
        ANALYZER FOR ASSIGNMENT STATEMENT             16 HELP∅∅8∅
ASST,   TO ASST2 IF (CHAR)='', TO EX.                 17 HELP∅∅8∅
ASST2,  H=CDR Y, USE FINPUT.                          18 HELP∅∅8∅
        TO ERR25 IF Z NE 19.                          19 HELP∅∅8∅
        PUSH DOWH T, CAR T=ASST1, TO REX.             2∅ HELP∅∅8∅
ASST1,  X=NEW ELEMENT.                                21 HELP∅∅8∅
        CAR X=L17, CDR X=CAR R.                       22 HELP∅∅8∅
        CAR R=X, POP UP T, TO CAR T.                  23 HELP∅∅8∅
ERR25,  1='2, 2='5, TO ERROR.                          2 HELP∅∅7∅
```

If a left part of an assignment statement is not identified, the recognized global variable is a primary. The expected construction is an expression and the control is transferred over to the label EX. Error 25 is signalled when an assignment statement is expected but the left-hand part global variable is not followed by an assignment operator.

For the execution of an assignment statement the following WISP lines are added:

```
.       EVALUATION OF AN ASSIGNMENT STATEMENT     1 HELP∅272
L17,    PUSH DOWN T, CAR T=L171.                  2 HELP∅272
        E=CDR E, TO CAR E.                        3 HELP∅272
L171,   CAR H=CAR R.                              4 HELP∅272
        POP UP T, TO CAR T.                       5 HELP∅272
```

The value assigned to a global variable can be an atom or a list. This value is obtained by evaluation of the right-hand side expression.

When a global variable is recognized as a primary, it must be analyzed by the analyzer for primaries. For this purpose the following lines are inserted:

```
        TO PR2∅ IF Z=2∅.                          1 HELP∅147
PR2∅,   Y=CDR Y, TO PR21 IF CAR Y NE NIL.         1 HELP∅158
        1='2, 2='6, TO ERROR.                     2 HELP∅158
PR21,   Y=CAR Y, USE GVCMP, TO PR3.               3 HELP∅158
```

Error 26 is printed out when a global variable used as a primary has an undefined value. The subroutine GVCMP serves for compilation of a global variable into the accepted intermediate form and could be considered as an analyzer for global variables. A special subroutine for evaluation of a global variable, used as a primary, is not necessary because the global variables are compiled as atoms or lists and they are evaluated by the appropriate subroutines.

3. **Deletion of global variables and function definitions.** During the execution of the user's programs some of the global variables could become not necessary any longer. Deletion of the name and the value of a global variable from the G-dictionary is provided by applying the subroutine DELETE. Deletion of a function definition is provided too, which permits to release memory when a function definition is no more necessary. The user might add to his program the following new HELP operators:

DEL global-variable-name;
DEL function-name.

After deletion of a function, it could be defined again. If the implementor wants to prohibit the redefinition of a function before the DEL operator is applied, the HELP∅∅55 line of the WISP text of the processor HELP must be changed:

```
        TO ERR19 IF CAR P NE NIL.                 X HELP∅∅55
```

For the deletion of a global variable and a function definition the following WISP operators are inserted:

```
        TO DLT IF Z=21.                           2 HELP∅∅5∅
DLT,    USE FINPUT, P=CDR Y.                      1 HELP∅∅8∅
        TO DLTF IF Z=∅3.                          2 HELP∅∅8∅
```

```
          TO ERR24 IF Z NE 2∅.                    3 HELP∅∅8∅
          D='G, TO DLTC.                          4 HELP∅∅8∅
DLTF,     D='F.                                   5 HELP∅∅8∅
DLTC,     USE DELETE, USE FINPUT.                 6 HELP∅∅8∅
          TO L∅1 IF Z NE ∅9, TO NPRINT.           7 HELP∅∅8∅
NPRINT,   (NIL)=∅1.                               X HELP∅∅77
ERR24,    1='2, 2='4, TO ERROR.                   1 HELP∅∅7∅
```

HELP∅∅77 line must be changed to provide printing of the deleted element name. When an attempt to delete another element except a global variable or a function definition is done, error 24 is printed out. To delete a function definition the following changes into the subroutine FINPUT are made:

```
          TO FUNC IF (CHAR)=';.                   1 HELP∅5∅5
BAKF,     Z=∅3, TO NOCH IF (CHAR)='(.             X HELP∅515
          EXIT FINPUT.                            1 HELP∅515
```

4. Segmentation and batch processing. An attempt to avoid the difficulties in the external memory usage in the list processors is carried out. A possibility for execution of large programs in a limited core memory only is proposed. The problem to be programmed could be divided into consistent parts. The transmission of data between them is to be performed by means of the introduced global variables. These parts could be considered as segments.

At the beginning of the HELP working the built-in functions must be saved. The stack N and subroutine STODEF are used for that purpose.

```
N=NEW ELEMENT, CAR N=NIL, CDR=NIL.    1 HELP∅∅12
USE STODEF.                           1 HELP∅∅19
USE STODEF.                           1 HELP∅∅24
USE STODEF.                           1 HELP∅∅29
USE STODEF.                           1 HELP∅∅34
USE STODEF.                           1 HELP∅∅39
```

A signal to open a new segment is the inserted into the user's program new HELP operator

```
SEGM;
```

SEGM is a reserved word. When it is identified, Z is set to 22 which is the internal number of that syntactical element. The control is transferred to the analyzer for seg nent.

```
          TO SEGMPR IF Z=22.                      3  HELP∅∅5∅
          ANALYZER FOR SEGMENT                    1∅ HELP∅∅8∅
SEGMPR,   TO L∅1 IF (CHAR) NE ';.                 11 HELP∅∅8∅
          F=NIL, W=N.                             12 HELP∅∅8∅
RESTOF,   USE FETDEF, TO RESTOF IF CAR W NE NIL.  13 HELP∅∅8∅
          Z='X, CAR Z='Y, A=NIL                   14 HELP∅∅8∅
          Y=G, USE DICSCAN, TO SEGMC.             15 HELP∅∅8∅
```

The function dictionary F is reset and the built-in functions only are copied into it again. This is performed by the subroutine FETDEF. The subroutine DICSCAN is used to scan the global variable dictionary G and to purge out the atom dictionary A. Only the atoms, used into the values of the global variables, are copied into the previously cleared A-dictionary. The control is

transferred to SEGMC and the execution of the next segment is initiated.

```
SEGMC,  Q=NEW ELEMENT, CDR Q=NIL.                    X HELP∅∅42
```

It is advisable to delete all unnecessary global variables before a segmentation call.

A possibility for a batch processing of user's programs is provided by the new HELP operator

```
PROG;
```

The names of the atoms TRUE and FALSE and the built-in functions must be defined at the beginning of every user's program as described in [1]. The card with these names must follow the card with the PROG operator.

The following additions and changes in the HELP processor are done:

```
        TO PROGPR IF Z=23.                           4 HELP∅∅5∅
.       ANALYZER FOR PROGRAM                         8 HELP∅∅8∅
PROGPR, TO L∅1 IF (CHAR) NE ';, TO PROGC.            9 HELP∅∅8∅
PROGC,  USE INITL.                                   X HELP∅∅∅1
```

5. Insertions into the reserved word dictionary. The reserved words DEL, SEGM and PROG are inserted into the reserved word dictionary BASIC. The following element definitions are used:

```
B7,     ELEMENT ∅∅ ∅∅ B71 B8.                        X HELP∅641
B8,     ELEMENT ∅∅ ∅∅ B81 B9.                        1 HELP∅641
B9,     ELEMENT ∅∅ ∅∅ B91 NIL.                       2 HELP∅641
B23,    ELEMENT ∅∅ ∅∅ B231 B24.                      X HELP∅648
B24,    ELEMENT ∅∅ ∅∅ B241 NIL.                      X HELP∅649
B231,   ELEMENT ∅∅ ∅∅ 'F B232.                       1 HELP∅649
B232,   ELEMENT ∅∅ ∅∅ NIL 12.                        2 HELP∅649
B241,   ELEMENT ∅∅ ∅∅ 'L B242.                       3 HELP∅649
B242,   ELEMENT ∅∅ ∅∅ NIL 21.                        4 HELP∅649
B81,    ELEMENT ∅∅ ∅∅ 'S B82.                        1 HELP∅673
B82,    ELEMEMT ∅∅ ∅∅ 'E B83.                        2 HELP∅673
B83,    ELEMENT ∅∅ ∅∅ 'G B84.                        3 HELP∅673
B84,    ELEMENT ∅∅ ∅∅ 'M B85.                        4 HELP∅673
B85,    ELEMENT ∅∅ ∅∅ NIL 22.                        5 HELP∅673
B91,    ELEMENT ∅∅ ∅∅ 'P B92.                        6 HELP∅673
B92,    ELEMENT ∅∅ ∅∅ 'R B93.                        7 HELP∅673
B93,    ELEMENT ∅∅ ∅∅ 'O B94.                        8 HELP∅673
B94,    ELEMENT ∅∅ ∅∅ 'G B95.                        9 HELP∅673
B95,    ELEMENT ∅∅ ∅∅ NIL 23.                       1∅ HELP∅673
```

6. Implementation of integer arithmetics. Non-numeric data processing problems often involve integer arithmetics to a certain extent. Thus, an implementation of integer arithmetics in HELP is necessary. It is carried out by means of built-in functions. The predicate INTEGER is used to check if a given atom is an integer. The one-argument function MINUS and the two-argument functions SUM, DIFFERENCE, PRODUCT, QUOTIENT and REMAINDER perform the arithmetic operations implicated by their names. Six built-in functions relating integers are added as well. They are the predicates EQ, NQ, GT, GE, LT and LE. Their values TRUE or FALSE could be used for conditional transfer of control.

An integer in HELP is any string of digits possibly beginning with a plus or a minus sign and always preceded by an asterisk. Examples of integers are *1976, *—796, *—∅∅∅5. The integers are atoms and are stored in the atom dictionary A. The plus and minus signs are inserted into the atom character set of the HELP processor. The identifier character set is not changed.

```
ATOM,   W='X, USE NEXTCH, USE READST.          X  HELP∅491
IDFN,   W=NIL, USE READST.                     X HELP∅5∅1
        TO IFID IF W NE 'X.                    1  HELP∅561
        TO LOOP IF (CHAR)='+.                  2  HELP∅561
        TO LOOP IF (CHAR)='-.                  3  HELP∅561
IFID,   TO LOOP IF (CHAR)=letordig.            X  HELP∅562
```

When an arithmetic built-in function is to be computed, its arguments are sent to the IOCS where the arithmetic operations are actually performed. The result computed is sent back. In the case of the implementation on the FACOM 230-45S the decimal arithmetic operations of the c mputer are used. So the restrictions which these operations impose on the operands are valid for the arguments of the implemented built-in functions. If an argument is not accepted, an appropriate message is printed out by the IOCS. This way of implementing integer arithmetics depends on the computer used, but there are many machine-independent aspects of the problem which will be specified on WISP.

The arguments and the results could be transferred between the HELP and the IOCS character by character through a buffer provided in HELP and accessible both from HELP and from IOCS. For that purpose the following lines are inserted in the HELP and WISP files.

```
BUF,      ELEMENT ∅∅ ∅∅ ∅∅ ∅∅.          1  HELP∅676
BUF EQU   ZBUF&                          1  WSPM∅223
```

In that case the storage area of the element BUF will be accessed from the IOCS under the name of ZBUF.

Several subroutines are added to the IOCS. The control is transferred to them by a WISP operator of a new type with an immediate parameter:

```
CALL &.
```

The CALL operator has to be translated as a subroutine call or as a return-jump instruction. This must be provided in the WSPM file.

The I CS subroutines SND and RCV are used for a character transfer between HELP and IOCS. The subroutine INDRESET is used to reset certain conditions in IOCS before sending an argument to IOCS or receiving a result from IOCS. In IOCS the arguments are transformed from internal HELP processor code to a representation suitable for execution of the desired operations. This is performed by the subroutines SHF1 and SHF2 for the first and the second arguments respectively. The operations are triggered by calls to appropriate subroutines in IOCS: MINUSF, SUMF, DIFF, PRODF, QUOTF, REMDF, EQF, NQF, GTF, GEF, LTF, and LEF. The results are transformed to internal HELP processor code before sending them back to HELP.

The HELP predicate INTEGER differs from the other functions and could be specified on WISP code only.

```
L16.    EVALUATE THE BUILT-IN FUNCTION INTEGER   1 HELP∅471
        Z=CAR R, POP UP R.                       2 HELP∅471
```

```
        TO L321 IF CAR R NE 'F.                          3 HELP∅471
        TO L142 IF AF Z=∅∅, Z=CDR Z.                     4 HELP∅471
        TO L162 IF CAR Z='+.                             5 HELP∅471
        TO L162 IF CAR Z='-.                             6 HELP∅471
        TO L142 IF CAR Z NE digit.                       7 HELP∅471
L161,   Z=CDR Z.                                         8 HELP∅471
L163,   TO L161 IF CAR Z=digit.                          9 HELP∅471
        TO L141 IF CAR Z=NIL.                           1∅ HELP∅471
        TO L142 IF CAR Z=character.                      11 HELP∅471
        Z=CAR Z, TO L163.                                12 HELP∅471
L162,   Z=CDR Z.                                         13 HELP∅471
L164,   TO L161 IF CAR Z=digit.                          14 HELP∅471
        TO L142 IF CAR Z=NIL.                            15 HELP∅471
        TO L142 IF CAR Z=character.                      16 HELP∅471
        Z=CAR Z, TO L164.                                17 HELP∅471
```

The insertions into the HELP processor for the other integer arithmetics functions follow:

```
L2∅F.   ONE-ARGUMENT FUNCTION MINUS                      18 HELP∅471
        Z=CAR R, POP UP R.                               19 HELP∅471
        TO ARGNUMB IF CAR R NE 'F.                      2∅ HELP∅471
        TO NONATOM IF AF Z=∅∅.                           21 HELP∅471
        USE SENDING, CALL SHF1.                          22 HELP∅471
        CALL MINUSF, USE RESRCV.                         23 HELP∅471
L21F.   BUILT-IN FUNCTION SUM                            24 HELP∅471
        USE ARGSND, CALL SUMF, USE RESRCV.               25 HELP∅471
L22F.   BUILT-IN FUNCTION DIFFERENCE                     26 HELP∅471
        USE ARGSND, CALL DIFF, USE RESRCV.               27 HELP∅471
L23F.   BUILT-IN FUNCTION PRODUCT                        28 HELP∅471
        USE ARGSND, CALL PRODF, USE RESRCV.              29 HELP∅471
L24F.   BUILT-IN FUNCTION QUOTIENT                      3∅ HELP∅471
        USE ARGSND, CALL QUOTF, USE RESRCV.              31 HELP∅471
L25F.   BUILT-IN FUNCTION REMAINDER                      32 HELP∅471
        USE ARGSND, CALL REMDF, USE RESRCV.              33 HELP∅471
L26F.   BUILT-IN FUNCTION EQUAL                          34 HELP∅471
        USE ARGSND, CALL EQF, TO LVRES.                  35 HELP∅471
L27F.   BUILT-IN FUNCTION NOT-EQUAL                      36 HELP∅471
        USE ARGSND, CALL NQF, TO LVRES.                  37 HELP∅471
L28F.   BUILT-IN FUNCTION GREATER                        38 HELP∅471
        USE ARGSND, CALL GTF, TO LVRES.                  39 HELP∅471
L29F.   BUILT-IN FUNCTION GREATER-OR-EQUAL              4∅ HELP∅471
        USE ARGSND, CALL GEF, TO LVRES.                  41 HELP∅471
L3∅F.   BUILT-IN FUNCTION LESS                           42 HELP∅471
        USE ARGSND, CALL LTF, TO LVRES.                  43 HELP∅471
L31F.   BUILT-IN FUNCTION LESS-OR-EQUAL                  44 HELP∅471
        USE ARGSND, CALL LEF, TO LVRES.                  45 HELP∅471
LVRES,  TO L141 IF (BUF)='1, TO L142.                    46 HELP∅471
NILVAL, 2='A, TO FAIL.                                   47 HELP∅471
NONATOM, 2='B, TO FAIL.                                  48 HELP∅471
NONINT, 2='C, TO FAIL.                                   49 HELP∅471
ARGNUMB, 2='D, TO FAIL.                                 5∅ HELP∅471
```

The HELP subroutine SENDING is used to send an argument of a built-in integer arithmetics function to the IOCS. The one-argument function MINUS uses SENDING directly. The HELP subroutine ARGSND is used to send the two arguments of the other built-in integer arithmetics functions to the IOCS. ARGSND uses SENDING as a subroutine. The HELP subroutine RESRCV is used to receive a result back from the IOCS.

The value set in the element BUF as a result of the execution of the built-in relational functions EQ, NQ, GT, GE, LT, and LE is one for TRUE and zero for FALSE.

Several interpreter errors can be recognized and indicated during an execution of an integer arithmetics built-in function.

ERROR A — An argument has a NIL value
ERROR B — An argument is a list
ERROR C — An argument is not an integer
ERROR D — Incorrect number of arguments

The approach towards the integer arithmetics implementation described does not imply principle changes in the original HELP processor. The execution time of the built-in functions is comparatively long, but the arithmetics facilities of the non-numeric processor HELP are to be considered auxiliary.

7. Subroutines added to the HELP system. The WISP text with comments of the subroutines added to the HELP system follows.

The subroutine NAMESCAN is used to copy into a linear list a name of a dictionary entry. On entry, the base register Y has to point to the first character of the name. On exit, the base register S will point to a linear list, representing the desired copy of the name. Both CAR and CDR fields of the last element of list S will contain NIL. X and Z are auxiliary pointers. The test for a letter, or a digit, or a space is represented by the incorrect WISP operator TO NLDS IF CAR Y=let-dig-sp.

```
        ENTRY NAMESCAN.
        S=NEW ELEMENT, Z=S.                 Form the list S
NSCAN,  TO NEND IF CAR Y=NIL.               Test for end of name
        TO NLDS IF CAR Y=let-dig-sp.        Test for let-dig-sp
        Y=CAR Y, TO NSCAN.                  No, it is a branch point
NLDS,   X=NEW ELEMENT.                      Yes, get new element
        CAR Z=CAR Y, CDR Z=X.               Copy a character
        Y=CDR Y, Z=CDR Z, TO NSCAN.         Prepare next checking
NEND,   CAR Z=NIL, CDR Z=NIL.               Set the last element
        EXIT NAMESCAN.
```

The subroutine STODEF is used to save a function definition. The subroutine FETDEF serves to restore a definition saved. N is a stack. Z and W are auxiliary pointers.

```
        ENTRY STODEF.
        PUSH DOWN N, CAR N=CAR Y.           Save the value
        Y=CDR Y, USE NAMESCAN.              Get the name
        PUSH DOWN N, CAR N=S.               Save the name
        EXIT STODEF.
        ENTRY FETDEF.
        Y=CAR W, USE NAMESCAN.              Get the name
```

```
          D='F, M='X, USE LOOKUP.        Lookup the F-dictionary
          Z=NEW ELEMENT, CDR Y=Z.        Get new element
          Y=CDR Y, CDR Y=S.              Restore the name
          W=CDR W, CAR Y=CAR W.          Restore the value
          W=CDR W, EXIT FETDEF.
```

The subroutines DICSCAN, LISTSCAN and ATOMSCAN serve to scan the G-dictionary and restore into the cleared A-dictionary all atoms which form the values of the global variables. This is necessary for the segmentation of the user's programs.

```
          ENTRY DICSCAN.
          TO DEXIT IF Y=NIL.             It the dictionary is empty
          X='Y.                          Set a fence
DLIST,    PUSH DOWN N, CAR N=X.          Save the return point
DLDS2,    TO DLDS1 IF CAR Y=let-dig-sp.  Check for let-dig-sp
          TO DVALUE IF CAR Y=NIL.        Check for end of list
          X=Y.                           It is a branch point
DNEXEL,   Y=CAR X, TO DLIST.             Move to the left entry
DLDS1,    Y=CDR Y, TO DLDS2.             Scan if let-dig-sp
DADV,     TO DPOPUP IF CDR X=NIL.        Check for completion of list
          X=CDR X, TO DNEXEL.            No, move to next element
DVALUE,   Y=CDR Y.                       Move to defining element
          TO DPOPUP IF CAR Y=NIL.        Check for undefined value
          Y=CAR Y, USE LISTSCAN.         Get the value
DPOPUP,   X=CAR N, POP UP N.             Restore the return point
          TO DADV IF AF X=∅∅.            Check for no fence
DEXIT,    EXIT DICSCAN.
          ENTRY LISTSCAN.
          TO LNOATOM IF AF Y=∅∅.         Check for atom
          USE ATOMSCAN, TO LEXIT.        Yes, use ATOMSCAN
LNOATOM,  X='Y.                          No, set a fence
LLIST,    PUSH DOWN N, CAR N=X.          Save the return point
          X=Y.                           It is a branch point
LNEXEL,   Y=CAR X.                       Move to the left entry
          TO LLIST IF AF Y=∅∅.           Check for no fence
          USE ATOMSCAN.                  Restore an atom
LADV,     TO LPOPUP IF CDR X=NIL.        Check for completion of list
          X=CDR X, TO LNEXEL.            No, move to next element
LPOPUP,   X=CAR N, POP UP N.             Restore the return point
          TO LADV IF AF X=∅∅.            Check for no fence
LEXIT,    EXIT LISTSCAN.
          ENTRY ATOMSCAN.
          Y=CDR Y, USE NAMESCAN.         Get the atom name
          D='A, M='X, USE LOOKUP.        Lookup the A-dictionary
          TO AEXIT IF CDR Y NE NIL.      Check for new entry
          Z=NEW ELEMENT, CDR Y=Z.        Yes, get new element
          AF Z=∅1.                       Set the atom flag
          X=NIL, CAR Z=CAR X.            Attach the base register
          CAR X=Z, CDR Z=S.              chain, set back pointer
AEXIT,    EXIT ATOMSCAN.
```

The subroutine DELETE performs a complete deletion of a dictionary entry. It deletes not only the value, but the name of the entry too.

```
          ENTRY DELETE.
          Z=S, W=D.                          Set travelling pointers
DNEXT,    TO DABS IF CDR D=NIL.              Check for dictionary end
          D=CDR D, Y=CAR D.                  No, work on next entry
DCHECK,   TO DNEXT IF CAR Y NE CAR Z.        Check for mismatch
          TO DDLT IF CAR Y=NIL.              Check for end of entry
DSTEP,    Y=CDR Y, Z=CDR Z.                  Step to next character
          TO DSUBCH IF CAR Y NE CAR Z.       Check for mismatch
          TO DSTEP IF CAR Y NE NIL.          Check for end of symbol
          TO DDLT.                           Delete it
DSUBCH,   D=Y, W=D, Y=CAR D.                 Check for subdictionary
          TO DCHECK IF AF Y=∅∅.              Yes, search it
DDLT,     TO DLT2 IF D NE W.                 It is first entry
          Y=CDR D.                           Move to next entry
          TO DCOM IF CDR Y NE NIL.           Only one more entry?
          Y=CAR Y.                           Yes, delete branch point
DCOM,     CAR D=CAR Y, CDR D=CDR Y.          No, retain branch point
DABS,     EXIT DELETE.
DLT2,     TO DLT3 IF D NE CDR W.             It is second entry
          TO DLT21 IF CDR D NE NIL.          Is it last entry?
          Y=CAR W, D=W, TO DCOM.             Yes, delete branch point
DLT21,    Y=CDR D, TO DCOM.                  No, retain branch point
DLT3,     TO DLT31 IF CDR D=NIL.             It is third or etc. entry
          Y=CDR D, TO DCOM.                  No last entry
DLT31,    W=CDR W.                           Last entry
          TO DLT31 IF CDR W NE D.            Scan the list
          CDR W=NIL, EXIT DELETE.            Delete last entry
```

The subroutine GVCMP is a compiler for global variables when they are recognized as primaries. During the compilation new structures are created and put into the result stack R.

```
          ENTRY GVCMP.
          TO GNOATOM IF AF Y=∅∅.             Test for atom value
          Z=NEW ELEMENT.                     Get new element
          CAR Z=L1, CDR Z=Y.                 Form an atom structure
          CAR R=Z, EXIT, GVCMP.              Include it into R stack
GNOATOM,  Z=NEW ELEMENT, CAR R=Z.            New element into R stack
          J=NEW ELEMENT.                     Get new element
          CAR Z=L2, CDR Z=J.                 Form a list structure
          PUSH DOWN S, CAR S='Y.             Set a fence
          I=J, X=Y.                          Set auxiliary pointers
GNEXEL,   Y=CAR X.                           Move onto the value
          TO GLIST IF AF Y=∅∅.               Test for atom
          Z=NEW ELEMENT.                     Yes, get new element
          CAR Z=L1, CDR Z=Y.                 Form an atom structure
          CAR I=Z.
```

```
GADV,    TO GENDLIS IF CDR X=NIL.        Test for end of list
         X=CDR X.                        No, work on next element
         Z=NEW ELEMENT.                  Get new element
         CDR I=Z, I=CDR I.               Set pointer to it
         TO GNEXEL.                      Process the element
GENDLIS, CDR I=NIL.                      Set end of list
         X=CAR S, POP UP S.              Restore X pointer
         TO GADV1 IF AF X=∅∅.            Check for a fence
         EXIT GVCMP.                     Yes, it is a fence
GLIST,   PUSH DOWN S, CAR S=I.           Save I pointer
         PUSH DOWN S, CAR S=X.           Save X pointer
         Z=NEW ELEMENT.                  Get new element
         J=NEW ELEMENT.                  Get new element
         CAR Z=L2, CDR Z=J.              Form a list structure
         CAR I=Z.
         I=J, X=Y, TO GNEXEL.            Work on next entry
GADV1,   I=CAR S, POP UP S, TO GADV      Restore I pointer
```

The subroutine SENDING is used to send an argument of an integer arithmetics built-in function to the IOCS.

```
         ENTRY SENDING.
         CALL INDRESET, Z=CDR Z.         Get first character
         TO S∅1 IF CAR Z='+.             Is it a plus sign
         TO S∅1 IF CAR Z='-.             Is it a minus sign
         TO S∅2 IF CAR Z=digit.          Is it a digit
         TO NONINT.                      No, arg. is not integer
S∅3,     TO S∅1 IF CAR Z=digit.          Check for digit
         TO S∅4 IF CAR Z=NIL.            Check for end of arg.
         TO NONINT IF CAR Z=character.   Argument is not integer
         Z=CAR Z, TO S∅3.                Advancing through CAR Z
S∅1,     (BUF)=CAR Z, CALL SND.          Send a character
         Z=CDR Z, TO S∅3.                Advancing through CDR Z
S∅2,     (BUF)='+, CALL SND, TO S∅1.     Send implicit plus sign
S∅4,     (BUF)=', CALL SND.              Send an ending space
         EXIT SENDING.
```

The subroutine ARGSND is used to send the two arguments of an integer arithmetics two-argument built-in function to the IOCS.

```
         ENTRY ARGSND.
         Z=CAR R, POP UP R.              Get the second argument
         TO ARGNUMB IF CAR R='F.         If only one argument
         TO NONATOM IF AF Z=∅∅.          If non-atom argument
         USE SENDING, CALL SHF2.         Send second argument
         Z=CAR R, POP UP R.              Get the first argument
         TO ARGNUMB IF CAR R NE 'F.      If more than two arguments
         TO NONATOM IF AF Z=∅∅.          If non-atom argument
         USE SENDING, CALL SHF1.         Send first argument
         EXIT ARGSND.
```

The subroutine RESRCV is used to receive a result of the execution of an integer arithmetics built-in function from the IOCS.

```
        ENTRY RESRCV
        CALL INDRESET, B=S, Z='S.          Save the S-list
R∅1,    CDR Z=NEW ELEMENT, Z=CDR Z.        Get new element
        CALL RCV, TO R∅2 IF (BUF)='.       Receive a character
        CAR Z=(BUF), TO R∅1.               Loop if not space
R∅2,    CAR Z=NIL, CDR Z=NIL.              Close new S-list
        D='A, M='X, USE LOOKUP.            Check the A-dictionary
        TO R∅3 IF CDR Y NE NIL.            Is there such integer, yes
        Z=NEW ELEMENT, CDR Y=Z.            No, attach new base reg.
        AF Z=∅1.                           Set the atom flag
        X=NIL, CAR Z=CAR X.                Attach the base register
        CAR X=Z, CDR Z=S.                  chain, set back pointer
R∅3,    CAR R=CDR Y, S=B, TO RET.          Restore the S-list, to RET
        EXIT RESRCV.
```

For the name definition of every one of the built-in functions added to the HELP system the following sequence of WISP operators must be inserted into the HELP file after the card with sequential number HELP∅∅4∅.

```
TO QUIT IF Z NE ∅3, X=NEW ELEMENT.
CAR X=label-of-evaluation-routine.
CDR X='F, Y=CDR Y, CAR Y=X.
USE STODEF, USE FINPUT.
```

The first card of the user's program must reflect the definition order, for example:

```
*T, *F, CAR(CDR(CONS(ATOM(NULL(I(M(SM(DF(PR(QT(RM(EQ(NQ(GT
(GE(LT(LE(;
```

8. Conclusions. — The modified HELP language resembles a conventional language. The flexibility of the system is increased and the programming of complicated problems is simplified.

— The introduced segmentation provides execution of large user's programs into a core memory limited in size. The utilization of external memory is avoided. Such a type of segmentation could be applied to other list processors as well.

— Batch processing of the user's programs is provided.

— The implemented integer arithmetics extends the application field of the HELP processor.

— The proposed modified HELP processor retains all possibilities of the original version described in [1]. It can be implemented without any efforts by anyone who is familiar with that original version.

— The extensions introduced increase the size of the processor program part with about 30 %. If a suitable overlaying of the processor programs is applied, the increase of memory needed will be considerably smaller.

REFERENCES

1. W. Waite. Implementing Software for Non-Numeric Applications. New York, 1973.
2. D. Bobrow. Symbol Manipulation Languages and Techniques. Amsterdam, 1968.

Ministry of Supply and State Resources
Computer Center *Sofia*

Received 12. 4. 1976