SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 1053–1069 (1981)

Experience with Software Conversion

HANAN SAMET

Computer Science Department, University of Maryland, College Park, Maryland 20742, U.S.A.

SUMMARY

Experience with a program to convert from LISP 1.6 to INTERLISP is described. The conversion program was designed with two goals in mind. First, it had to be capable of being executed in either of the languages's environments and it had to yield identical results. Second, the speed of the converted program was to be approximately the same as the original program. This meant that the conversion process must be completed prior to execution of the converted program. The various constraints and considerations imposed by these goals are examined. In addition, aside from problems in finding INTERLISP analogs for various LISP 1.6 constructs, careful consideration must also be paid to input/output functions, escape characters, global variables, representation of numbers and different string implementations.

KEY WORDS Software conversion Bootstrapping LISP INTERLISP Portable software Program transformation

1. INTRODUCTION

The tendency to do as much programming in a high level language as possible has been influenced to a high degree by the desire to avoid having to reinvent the wheel. In particular, a goal is to have portable software. This has led to standardization efforts such as Reference 1. Nevertheless, with the proliferation of programming languages and computers rare is the language whose programs are really transportable. A common problem is that although there is a general adherence to the kernel of the language, successive implementations rarely resist the temptation to enhance the previous version. The result is that although the various implementations of the language are structurally the same; enough incompatibilities have been introduced so that often programs written in one implementation will not run in the other. This is taken for granted when one implementation is a superset of the other—i.e. programs written in the superset will not run in the subset. However, more often than not, programs written in the subset will also not run in the superset.

Some of the main reasons for incompatibility are changes in the input/output structure, data structure definitions, and also in the degree of run-time support. Most often, initial implementations are lacking in sophisticated input/output capabilities. Subsequently, the language is enhanced. This is frequently achieved by adding built-in functions which make the programmer's task easier. Since input/output is often not defined for a language (e.g. ALGOL60,[2] LISP[3]), compatibility in these cases has not been a burning issue. Nevertheless, a program is generally worthless if it cannot do input or output. Thus there is compatibility as long as no meaningful work needs to be done. This is unfortunate because, for most programming languages, input/output is

0038–0644/81/101053–17$01.70

Received 19 March 1979

one of the most difficult concepts to grasp (it is also often the last concept that is mastered and, in fact, most programmers have little mastery above the basic rudiments). Therefore, the programmer is confronted with the burden of incompatibility in the part of the language with which he is the least familiar.

This paper arose out of an effort to convert a complex program used in a compiler testing system[4] written in LISP 1.6[5] to INTERLISP.[6] INTERLISP and LISP 1.6 are both implementations of LISP and neither can be termed a subset or superset of the other. Nevertheless, it is generally acknowledged that, in an inexact manner of speaking, LISP 1.6 is more of a subset of INTERLISP than vice versa.*

INTERLISP is an enhancement of LISP which supports a full complement of debugging aids and input/output facilities. The documentation also claims the existence of an INTERLISP program named TRANSOR which contains, among others, transformations to convert from LISP 1.6 to INTERLISP. Unfortunately, attempts to use this set of functions were unsuccessful because only relatively primitive patterns were included, and no capabilities for transforming the input/output functions were provided. Thus we were compelled to undertake this study.

In the remainder of this paper we discuss some of the important considerations which we needed to take into account in order to convert the existing programs. In particular, one of the primary goals was that the conversion program be capable of being executed in either a LISP 1.6 or an INTERLISP environment. Thus the conversion program had to be able to convert itself (i.e. bootstrapping). A second goal was that the conversion process be completed prior to executing the converted program. These considerations forced changes in the original source programs by specifying some restrictions on program structure although they did not hamper the flexibility of the original program. Instead, they appear to be common to convertible software and thus are worthy of discussion. Once this is done, the encoding of the conversion program is presented using an ALGOL-like variant of LISP.†

2. CONVERSION CONSIDERATIONS

As mentioned in Section 1, many of our design decisions were motivated by the requirement that the conversion be capable of taking place in either language environment without any user intervention. This will be seen to heavily influence many of our design decisions. We also take advantage of a common feature of many LISP implementations which allows functions to be redefined. In essence, if a definition of a previously defined function is encountered when loading a set of function definitions, then the old definition is overwritten. We make heavy use of this property by appending to every translated program a number of INTERLISP function definitions for certain primitive constructs useful in input/output operations. In the remainder of this section we discuss the conversion of standard functions and file input/output, escape character, strings, unique name generation, numbers, and global variables.

---

*A more precise term to describe the relationship between the two implementations is that one is a sideset of the other.[7]

†This is commonly referred to as meta-lisp or blackboard LISP and is very similar to MLISP.[8]

2.1. Standard functions

Most LISP 1.6 built-in functions have analogs in INTERLISP which are either identical in structure (i.e. in the number and relative position of the arguments) or are very similar so that a simple rearrangement of arguments and renaming are all that is necessary for conversion. Nevertheless, a number of LISP 1.6 functions are defined differently in INTERLISP.* For example, PRINT(X) in LISP 1.6 prints a carriage return followed by X while in INTERLISP X is printed before the carriage return. As another example, the SUBST function is different. In LISP 1.6, SUBST(NEW,OLD,SEXPR) results in an attempt to substitute NEW for OLD in SEXPR and recursively in CAR and CDR of SEXPR, whereas in INTERLISP substitution only takes place when OLD is EQUAL to CAR of some subexpression of SEXPR. Therefore, in INTERLISP SUBST(A, (B C),((B C) D B C)) is equal to (A D B C) instead of (A.(D . A)) as in LISP 1.6.

There are two possible methods of coping with the differences in the definitions of the built-in functions. The first is to redefine the INTERLISP function to correspond to the LISP 1.6 definition. The problem with such an approach is that this changes the definition of the function in so far as the runtime system of INTERLISP is concerned. The second solution, which we adopt, is to change the names of these functions to names that do not have an analog in INTERLISP (e.g. PRINT to PRINT1) and append definitions for these functions to the converted program. We also append definitions for LISP 1.6 functions which have no INTERLISP analogs at all (e.g. XCONS, NCONS, ROUND, etc.).

A basic principle underlying our conversion system is that all conversion be completed prior to executing the converted program. Thus the conversion program is not present at runtime. This results in a number of restrictions. First, no functions may be defined 'on the fly'—i.e. at runtime and thus, also, they may not be redefined. Second, functional arguments must be used with caution. They will most likely require special handling at runtime since often they are used under the assumption that they will be evaluated in a LISP 1.6 environment. We have taken the step to convert all arguments to a FUNCTION construct (e.g. (FUNCTION *fname*)) to their INTERLISP analogs at conversion time. Such a treatment coupled with the fact that function redefinitions have been appended to the converted programs will have the desired effect in most cases. Nevertheless, problems may arise. This is especially true when functional forms appear as arguments to QUOTE. In such a case, we perform no conversion since we have no means of distinguishing between the prevention of evaluation for output (e.g. 'BLUE) and delayed evaluation (e.g. '(PLUS A B)). Of course, there are two alternatives. One is to perform conversion at runtime. However, this is too slow to be acceptable. A second possibility is to change EVAL of INTERLISP to correspond to that of LISP 1.6. However, this has the effect of precluding the use of many of INTERLISP's desirable features which are often encoded using the more basic functions.

Once the program has been converted, it is ready for execution. At this point the user must bear in mind that he is communicating with INTERLISP's EVAL. This means that any sexpressions which are typed are interpreted in the context of INTERLISP. For example, PRINT(X) will result in X being output followed by a carriage return rather than *vice versa*. However, in general, if the sexpression does not

*All LISP 1.6 constructs are handled with the exception of LEXPR, macros, and arrays. The only LISP 1.6 functions that have not been converted are BOOLE, *FUNCTION, TIME, *RSET, BAKGAG, INITFN, ARRAY, EXARRAY,STORE, EXAMINE, DEPOSIT, DDTIN and MAKNUM.

have a meaning in INTERLISP, then its LISP 1.6 definition will be invoked provided it exists (e.g. XCONS, NCONS, ROUND, etc.). Similarly, the fact that one is communicating with INTERLISP also means that in case of errors, different results may occur. For example, in INTERLISP, CAR of an atom is NIL whereas in LISP 1.6 two successive CAR operations applied to an atom result in an error message of the form 'ILLEGAL MEMORY REFERENCE FROM CAR'.

2.2. Input/Output

LISP 1.6 input/output is based on the concept of a channel while INTERLISP input/output is based on files. Prior to any LISP 1.6 input or output operations with respect to a file, the file must be associated with a channel (commonly termed initiating or opening a channel). The conversion program transforms the channel names into variables whose values are the file names. This is accomplished by generating global variable declarations for the channel names as well as initializing them to NIL. Thus the act of associating a channel with a file is transformed to an assignment of the file name as the value of the channel name. This insures that subsequent input/output operations that refer to the channel will access the appropriate file. In addition, we stipulate that the actual association of a channel with a file be done by use of the functions INITIALIZECHANNELFORINPUT and INITIALIZECHANNELFOROUTPUT which are analogous to the LISP 1.6 INPUT and OUTPUT functions.* These two routines have the following responsibilities:

1. Create a valid file name.
2. Open the file for input or output.
3. If the selected channel was previously open for input or output, then release (i.e. close) the file previously associated with it.
4. Store the name of the file as the value of the channel.

The LISP 1.6 definitions of these functions must be supplied by the programmer in his LISP 1.6 program while the conversion program appends the following INTERLISP equivalents (and the auxiliary functions MAKEFILENAME and INITCHAN) to the converted program. This has the effect of overrriding the earlier LISP 1.6 definitions of these functions. Also observe that these conventions only apply to file input/output since teletype input/output does not require initialization of channels.

```
(DEFINEQ (MAKEFILENAME                (* if a non-atomic name, then convert to )
      (LAMBDA (NAME)                  (* an atom comprising the name followed by)
        (COND ((ATOM NAME) NAME) (* a dot followed by the extension)
              (T (PACK (APPEND (UNPACK (CAR NAME))
                               (APPEND (LIST (FCHARACTER 46))
                                       (UNPACK (CDR NAME)))))))))

(DEFINEQ (INITIALIZECHANNELFORINPUT
      (LAMBDA (CHANNEL FILE)
        (INITCHAN CHANNEL FILE (QUOTE T)))))
```

*We also require that the arguments to INITIALIZECHANNELFORINPUT and INITIALIZECHANNELFOROUTPUT be quoted names and not formal parameter names.

```
(DEFINEQ (INITIALIZECHANNELFOROUTPUT
          (LAMBDA (CHANNEL FILE)
            (INITCHAN CHANNEL FILE NIL))))

(DEFINEQ (INITCHAN
          (LAMBDA (CHANNEL FILE INPT)
            (PROG (EVCHAN)
                  (SETQ FILE (MAKEFILENAME FILE))
                  (COND INPT (INFILE FILE))        (* open file )
                        (T (OUTFILE FILE)))
                  (SETQ EVCHAN (EVAL CHANNEL))  (* file associated with channel)
                  (COND ((NULL EVCHAN) NIL)              (* close file if one is)
                        ((OPENP EVCHAN) CLOSEF EVCHAN)) (* already open on )
                        (T NIL))                         (* the channel)
                   (SET CHANNEL FILE)))))          (* assign file to channel )
```

Once a file has been opened, LISP 1.6 requires that all subsequent input and output operations to and from the file must be preceded by an INC and OUTC respectively to select the file. The same operation is also used to close a file currently selected for input or output while simultaneously selecting another file for subsequent input or output. In such a case, the second argument to INC or OUTC is non-NIL and thus the conversion program must also generate a call to a function, CLOSEF, which closes the currently selected file.

In the case of input, frequently an end of file needs to be examined. This is a system-dependent construct and we stipulate that it be done by use of the following function call or equivalent:

```
(ENDOFFILECHECK (ERRSET (READ )))
```

Note the use of (ERRSET (READ)) as an argument. The ENDOFFILECHECK function must be supplied by the user in his LISP 1.6 program. The conversion program appends the following INTERLISP definition to the converted program:

```
(DEFINEQ (ENDOFFILECHECK)
         (LAMBDA (ITEM)
           (NULL ITEM))))
```

2.3. Escape character

In most programming languages certain symbols or sequences of symbols have predefined meanings. For example, in ALGOL 'begin' is a reserved word. Similarly, the doublequote symbol is used in many programming languages to delimit strings. In such a case, in order to allow a string to contain the doublequote symbol, the convention is adopted that inside a string, two successive doublequote symbols are interpreted as representing one doublequote symbol serving as a component of the string rather than denoting its end. In LISP, there are several symbols with predefined meanings in addition to doublequote. In particular, left and right parentheses must be marked in a special manner when they do not have their customary start and end of list meanings. The symbol used to achieve the marking is known as an escape character and in LISP 1.6 the slash symbol has this meaning. Thus whenever a slash symbol is encountered in a LISP 1.6 environment, the

immediately following character is taken as an ordinary character without its predefined meaning. In order to be able to use the slash symbol with its conventional meaning, it must be doubled. INTERLISP uses the percent symbol as its escape character and thus there is a possible incompatibility with LISP 1.6.

Fortunately, INTERLISP has a builtin function called SETSYNTAX. SETSYNTAX(CHARACTER, CLASS, TABLE) is a directive to the INTERLISP file input routines to treat CHARACTER as a member of character class CLASS when an input operation makes use of character table TABLE for character disposition. Our goal is to enable the slash symbol as the escape character upon input and to disable the percent symbol as the escape character upon input. This is achieved by the following two commands:

```
(SETSYNTAX (FCHARACTER 37) (QUOTE OTHER) FILERDTBL)
(SETSYNTAX (FCHARACTER 47) (QUOTE ESCAPE) FILERDTBL)
```

Both commands make use of table FILERDTBL which is a system defined breaktable for use upon file input/output. The first command disables the percent symbol (i.e. ASCII 37) as the escape character upon input while the second command enables the slash symbol (i.e. ASCII 47) for the same purpose. Note the use of the builtin function FCHARACTER with a numeric argument equal to the ASCII code of the character. This is preferable to using the character itself due to a possible special meaning within LISP 1.6. For example, if the original conversion program was written in MLISP, then the percent symbol serves to delimit comments and it is impossible to use it otherwise in the program.*

When the conversion program is executed in LISP 1.6, every converted file must be prefaced with the above pair of SETSYNTAX commands to enable and disable the slash and percent symbols, respectively, as the escape characters. This is necessary for the converted program to be loaded properly. If the converted program will be processing input that has been generated by LISP 1.6 (this is a stronger requirement than that the input be written in LISP 1.6), then once again the slash and percent symbols must be enable and disabled, respectively, as the escape characters. Note that if the input will not be written by LISP 1.6, then the percent and slash symbols should be enabled and disabled, respectively, as the escape characters. This is achieved by the following SETSYNTAX commands:

```
(SETSYNTAX (FCHARACTER 47) (QUOTE OTHER) FILERDTBL)
(SETSYNTAX (FCHARACTER 37) (QUOTE ESCAPE) FILERDTBL)
```

The conversion program could also be written in such a way that the percent symbol is always used as the escape character regardless of whether the conversion takes place in a LISP 1.6 or INTERLISP environment. In other words, there is never a need to preface the converted file with the SETSYNTAX commands that change the escape character. However, this technique slows down the conversion process considerably since it requires that output of the resulting INTERLISP program be done on a character by character basis. This is necessary to insure that each slash symbol is replaced by a percent symbol except when it is preceded by a slash. Similarly, all uses of the percent symbol must be duplicated.

Since the escape character is different for the two LISPs, all references to it are via the function PRINCESCAPECHARACTER() whose value is the current escape character. In

* The inability to disable this effect is clearly a design flaw of MLISP.

order for the INTERLISP version of the conversion program to function properly, PRINCESCAPECHARACTER is redefined to yield the percent symbol by appending the following definition to every converted program:

```
(DEFINEQ (PRINCESCAPECHARACTER
          (LAMBDA NIL
          (FCHARACTER 37))))
```

An alternative solution is to use a global variable, e.g. PRINCESCAPECHARACTERG. Unfortunately, this solution does not work because it is difficult to override the original assignment of the slash symbol to the variable when executing in LISP 1.6. Note that use of a function causes such a problem to disappear since a function redefinition is taking place when the inserted PRINCESCAPECHARACTER function definition is encountered.

2.4. Strings

Strings are defined somewhat differently in INTERLISP than in LISP 1.6. First, in INTERLISP a string is not an atom. This could prove troublesome for programs which rely on the fact that ATOM(X) returns true when X is an atom. One possible remedy is to transform all tests of the form (ATOM A) to (OR (ATOM A) (STRINGP A)). The only problem with such a transformation is that the computation corresponding to A may have side-effects and thus should not be performed twice. Therefore, a preferable transformation (and the one we use) changes all uses of ATOM to ATOM1 which has the following definition:

```
(DEFINEQ (ATOM1 (LAMBDA (X) (OR (ATOM X) (STRINGP X)))))
```

Second, the definition of the characters comprising a string is different in the two implementations. In particular, LISP 1.6 does not treat the slash symbol as an escape character within a string, preferring to use two doublequote symbols when the doublequote symbol appears as part of a string. On the other hand, INTERLISP is consistent in this respect and continues to treat the percent symbol as an escape character.

The different definitions of the characters comprising a string present a number of problems for the conversion procedure. First, the disabling of the slash symbol as an escape character in LISP 1.6 strings means that conversions performed in a LISP 1.6 environment will not be identical to conversions in INTERLISP. A possible remedy is to modify the LISP 1.6 conversion output routine to check every list element to determine if it is a string. In the affirmative case, the individual characters of the string must be checked for the presence of the slash symbol, and all such occurrences must be converted to two consecutive slash symbols since the INTERLISP file input routine has been primed to treat the slash symbol as an escape character. Similary, in LISP 1.6, all occurrences of a pair of consecutive doublequote symbols within a non-empty string must be replaced by a single slash symbol followed by a doublequote symbol. However, this solution is only adequate as long as the conversion program is not being executed in an INTERLISP environment. In the latter case, any occurrence of the slash symbol will be viewed by the file input routine as an escape character for the following character and thus will not be seen by the conversion program. In the case of two consecutive occurrences of the doublequote symbol, the problems are more serious. INTERLISP's file input routine will treat the first doublequote symbol

as terminating the current string while the second doublequote symbol will be interpreted as starting a new string.

Clearly, our goal of compatibility between the two systems is impossible to satisfy for strings containing the doublequote or slash symbols. Therefore, we stipulate that string constants, in order to be properly converted, must not contain the doublequote or slash symbols.

2.5. Atom names

In LISP 1.6 all identifiers that are not created by the program (i.e. via a READ) are hashed and placed on a list termed the OBLIST. This insures that there is only one atom associated with each identifier. This is not true for identifiers generated by GENSYM or MAKNAM.* If it is desired that such atoms be unique and therefore be present on the OBLIST, then an INTERN operation is performed. Atoms are removed from the OBLIST by use of the REMOB operation. In INTERLISP, all atoms are placed on the hash list and thus there is no need for the INTERN or REMOB operations. Therefore, the conversion program replaces all instances of INTERN and REMOB by their arguments.

Both LISP 1.6 and INTERLISP provide a capability for generating unique names via the GENSYM function. In LISP 1.6, the function CSYM, whose argument is a character sequence comprising one or more alphabetic symbols, say ABC, followed by a sequence of one or more digits, is used to indicate the form of the uniquely generated name. All subsequent names are obtained via the function GENSYM having zero arguments. Each time GENSYM is called, a name is generated having a numeric component one higher than the previously generated name. For example, a GENSYM following CSYM(ABC002) results in the name ABC003.

INTERLISP does not have an analog to CSYM. Instead, there exists a global variable called GENNUM whose value is the number associated with the most recently generated identifier. GENSYM is defined with one argument which is a character, and each call to GENSYM results in a name of the form X*nnnn* where X is the character argument to GENSYM and *nnnn* is a four digit number whose value is determined by adding one to the previous value of GENNUM.

The conversion program decomposes the argument to each CSYM into an alphabetic component and a numeric component. We make the restriction that only the first letter of the alphabetic component is used in the INTERLISP analog of the uniquely generated identifier. We use a global variable GENSYMCHAR to keep track of the current leading letter of uniquely generated identifiers.† All instances of (GENSYM) are replaced by (GENSYM GENSYMCHAR). An assignment statement is also generated to set GENNUM to the numeric component.

2.6. Numbers

In general, LISP 1.6 outputs all integer numbers in base-8. Therefore, when attempting to convert a program that has been written by LISP 1.6 (e.g. a LISP 1.6 program obtained by use of MLISP) then the output of the conversion must be a base-10 number. In fact, the only time that LISP 1.6 leaves numbers alone is when they appear within string constants. Therefore, numbers appearing within QUOTE and EVAL must also be converted to base-10. Note that there is a distinction between

---

*MAKNAM takes a list of characters and forms an identifier from them.

†Therefore, internally generated sequences of names should use different first letters.

programs written in LISP 1.6 (i.e. by the user) and programs written by LISP 1.6. In the former case, numbers are in base-10 and need no special handling. In the latter case, conversion is a necessity. The actual conversion is performed while the converted program is being output.

A LISP 1.6 program may override the default base of the output. This is achieved by setting the global variable BASE to the desired base (it is 8 by default). We achieve this by the following statement:

```
IF PRINCESCAPECHARACTER( ) EQ LISP16ESCAPECHARACTER THEN BASE : = 7+3;
```

Two items are worthy of note, First, the escape character has been used to detect the LISP environment in which the conversion program is being executed. Second, 7+3 is employed rather than 10 since it is a base-independent* method of obtaining the number 10 in base 10.

There is also a difference between the LISP 1.6 and INTERLISP representation of small integers. All integers having an absolute value of less than 2**16 are represented uniquely in LISP 1.6 whereas for INTERLISP this is only true for integers ranging between −1536 and +1535. Thus tests for equality between numbers in LISP 1.6 which make use of EQ must use EQP (an INTERLISP predicate for checking equality of numbers). Since LISP is typeless, we can't, in general, recognize such cases. Therefore, we change all instances of EQ to invoke EQ1 which has the following definition:

```
(DEFINEQ (EQ1 (LAMBDA (A B) (OR (EQ A B)(AND (FIXP A) (FIXP B))(EQP A B)))))
```

This has a further implication on builtin functions which make use of EQ. In particular, uses of ASSOC in LISP 1.6 are transformed to SASSOC in INTERLISP since the latter makes use of the more general EQUAL predicate rather than EQ. Note that similar problems may arise when using GET and MEMQ in LISP 1.6 and their corresponding INTERLISP analogs GETLIS and MEMB. However, in this case we do not provide any relief.

2.7. Global variables

In addition to the standardized input/output channel initialization, end of file check, and escape character usage we also require that all global variables (SPECIAL in the case of LISP 1.6) be declared prior to the set of functions which are being converted. These declarations are processed to yield a list of the form:

(ADDTOVAR GLOBALVARS ⟨*rest of global variable names*⟩)

3. CONVERSION FUNCTION

The conversion functions are presented below in a variant of MLISP.† Briefly, MLISP is an ALGOL-like variant of LISP whose biggest virtue is the ability to use 'if then else' and functional notation rather than COND and cambridge prefix notation. The value of a block is the value of the last statement. The construct NEW is used to allocate variables which are equivalent to PROG variables. ⟨A, B, C⟩ denotes the list

*This is true for base greater than or equal to 8. True base independence would be achieved by writing 1+1+1+1+1+1+1+1+1+1.

†Note the use of vertical bars to indicate the nesting of the blocks.

consisting of the elements A, B and C. Brackets of the form A[2,1] correspond to a sequence of CDR and CAR operations to extract list elements—e.g. A[2,1] is the first element of the second sublist. We have attempted to present most of the LISP 1.6 builtin functions. Nevertheless, it should be clear from the following how to add any functions that may have been omitted.

Procedure CONVERT is invoked with a LISP 1.6 function definition of the form:

(DEFPROP *fname* (LAMBDA *parameterlist functionbody*) *functiontype*)

functiontype is either EXPR or FEXPR corresponding, respectively, to the cases when the arguments are evaluated and when they are not. The actual conversion is done by the recursive procedure SUBSTITUTE which results in the transformation of functionbody from LISP 1.6 to INTERLISP. SUBSTITUTE makes use of the auxiliary function SUBSTITUTELIST when lists of items are to be converted. Procedure CHECKCHANNEL generates code to initialize a channel variable to NIL if it has not been encountered before. Channel variables are detected by examining the arguments to INITIALIZECHANNELFORINPUT and INITIALIZECHANNELFOROUTPUT. The global variable SEENCHANNELS is used to keep track of all channel names so that each channel variable is only initialized once.

In the conversion process we make use of the following lists of function names. INTERLISPANALOGS is a list of the following name pairs where the first element is the LISP 1.6 function while the second element is the corresponding INTERLISP function. Note that some of the INTERLISP functions have their definitions appended to the converted program (see the appendix for these definitions) while others are already present in INTERLISP.

*APPEND	APPEND
*DIF	*DIF
*GREAT	GREATERP
*LESS	LESSP
*PLUS	PLUS
*QUO	QUOTIENT
*TIMES	TIMES
ASCII	CHARACTER
ASSOC	ASSOC1
EXPLODEC	UNPACK
FLATSIZEC	NCHARS
GET	GET1
GETL	GETLIS
LAST	FLAST
MAKNAM	PACK
PRIN1	PRIN2
PRINC	PRIN1
READLIST	PACK
MAP	MAP1
MAPC	MAPC1
MAPLIST	MAPLIST1
MAPCAR	MAPCAR1
TERPRI	TERPRI1
PROG2	PROG3

SUBST	SUBST1
PRINT	PRINT1
ATOM	ATOM1
EQ	EQ1
GENSYM	GENSYM1
PUTPROP	PUTPROP1
ZEROP	ZEROP1
READ	READ1
SASSOC	SASSOC1
MEMQ	MEMB
LSH	LLSH
READCH	READC

IOFNS is a list of functions whose arguments are used to determine channel names. It consists of INITIALIZECHANNELFORINPUT and INITIALIZECHANNELFOROUTPUT. PREDICATES consists of the LISP 1.6 functions AND, NUMBERP, OR, REMPROP, MEMBER and MEMQ which have the distinction that their INTERLISP analogs do not always return a value of T when the result of their application is non-NIL as is required by LISP 1.6. The conversion process assures that T is returned in the appropriate case by appending a NULL test. Other functions which are singled out for special handling include DIFFERENCE, QUOTIENT, GREATERP and LESSP all of which take an arbitrary number of arguments and do not have a predefined INTERLISP analog.

```
expr CONVERTFN(FNDEF);
begin
  new BODY;
  BODY : = SUBSTITUTE(FNDEF[3,3]);
  print(〈'DEFINEQ,〈FNDEF[2],
                    if FNDEF[4] eq 'EXPR then 〈'LAMBDA,FNDEF[3,2],BODY〉
                    else 〈'NLAMBDA,
                         if null FNDEF[3,2] then 'NIL
                         else 〈FNDEF[3,2,1]〉,
                         BODY〉〉〉);
end;
expr SUBSTITUTELIST(SLIST);
mapcar(function(SUBSTITUTE), SLIST);
expr SUBSTITUTE(FORM);
begin
  new FNAME,ANALOGS,ARGS;
  if atom(FORM) then FORM
  else if atom(FORM[1]) then
    begin
      FNAME : = FORM[1];
      if FNAME eq 'COND then mapcar(function(SUBSTITUTELIST, cdr(FORM))
      else if FNAME eq 'PROG then
        'PROG cons FORM[2] cons mapcar(function(SUBSTITUTE), cddr(FORM))
      else if FNAME eq 'QUOTE then FORM
      else if FNAME eq 'FUNCTION then
```

```
        begin
          ANALOGS : = assoc(FORM[2],INTERLISPANALOGS);
          if ANALOGS then ⟨'FUNCTION,cdr(ANALOGS)⟩
          else FORM;
        end
      else
        begin
          ANALOGS : = assoc(FNAME,INTERLISPANALOGS);
          ARGS : = mapcar(function(SUBSTITUTE), cdr(FORM));
          if ANALOGS then cdr(ANALOGS) cons ARGS
          else if member(FNAME,PREDICATES) then
            ⟨'NULL,⟨'NULL,
                    (if FNAME eq 'MEMQ then 'MEMB
                     else FNAME)
                    cons ARGS⟩⟩
          else if member(FNAME,IOFNS) then CHECKCHANNEL(FNAME,ARGS)
          else if FNAME eq 'INC or FNAME eq 'OUTC then
            ⟨FNAME,
              if atom(ARGS[1]) or ARGS[1,1] neq 'QUOTE then ARGS[1]
              else ARGS[1,2],
              ARGS[2]⟩
          else if FNAME eq 'DIFFERENCE then
            ⟨'DIFFERENCE,ARGS[1],'PLUS cons cdr(ARGS)⟩
          else if FNAME eq 'QUOTIENT then
            ⟨'QUOTIENT, ARGS[1],'TIMES cons cdr(ARGS)⟩
          else if FNAME eq 'GREATERP then ⟨'GREATERP1, ARGS⟩
          else if FNAME eq 'LESSP then ⟨'GREATERP1, reverse(ARGS)⟩
          else NIL
        end;
    end
  else if FORM[1,1] eq 'LAMBDA then
    ⟨'LAMBDA,FORM[1,2],SUBSTITUTE(FORM[1,3])⟩ cons
    mapcar(function(SUBSTITUTE),cdr(FORM))
  else print(' ''ILLEGAL FORM'')
end;
expr CHECKCHANNEL(FNAME, ARGS);
begin
  if not member(ARGS[1,2],SEENCHANNELS) then
    begin
      SEENCHANNELS : = ARGS[1,2] cons SEENCHANNELS;
      print(⟨'SETQ,ARGS[1,2],NIL⟩);
    end;
  FNAME cons ARGS;
end;
```

In general, it is useful to distinguish between those conversions which deal with the external form of the program (e.g. whether strings contain double quotes or escape characters) and those that deal with the semantics of the basic constructs (e.g. different

functions for small integers, SUBST PRINT, etc.). It can be argued that the former could be more conveniently handled by a text processor. Similarly, many of the latter could be dealt with by a structure editor (e.g. permuting the arguments to MAP1). Our system is an attempt to deal with the conversion problem in a unified manner without sacrificing efficiency.

4. CONCLUSION

Clearly much of our discussion with respect to conversion problems has been heavily dependent on LISP. Nevertheless , a number of useful ideas were presented which should be applicable in other conversion tasks. For example, the escape character is a useful device for detecting the system in which execution is occurring, function redefinition is a powerful tool for overriding previous applicable function definitions, and standardized input/output functions are desirable in all programming languages. Moreover, INTERLISP's SETSYNTAX capability is an essential mechanism for resolving scanner problems* although we did discover some shortcomings. In particular, there was no way to declare that certain characters or sequences of characters are to be ignored upon input—i.e. omitted. This drawback surfaced when the conversion program was confronted by the sequence control-z, carriage return, and linefeed which was output by LISP 1.6 whenever a line of output was too long thereby requiring that an atom or string be output on more than one line. This problem was resolved by ensuring that LISP 1.6 outputs only one atom or string per line.

There were also a number of irreconcilable problems. This was exemplified by different definitions of data types (e.g. STRING and small integers). Thus it is clear that the issue of conversion is deeper than mere pattern matching. It also requires careful scrutiny of type incompatibilities. Moreover, in the interest of efficiency we deemed that all conversion must take place prior to executing the converted program. This means that certain features of LISP such as the ability to define and redefine functions 'on the fly' had to be sacrificed. Finally, we observe that program conversion also requires the imposition of a certain degree of discipline on programmers. However, we generally do not find such requirements to be unduly burdensome since they are often associated with good programming habits.

ACKNOWLEDGEMENTS

I have benefited greatly from discussions with Pete Alfvin , Steve Crocker, and Larry Musinter.

APPENDIX—FUNCTION DEFINITIONS

```
(DEFINEQ (INC
           (LAMBDA (A B)
             ((LAMBDA (X)
                (COND ((NULL B) (INPUT X))
                      (T (CLOSEF (INPUT X)))))
                (COND ((NULL A) T) (T A))))))
```

*SETSYNTAX uses a breaktable which is a consept that is also present in SAIL.[9]

```
(DEFINEQ (OUTC
           (LAMBDA (A B)
             ((LAMBDA (X)
                (COND ((NULL B) (OUTPUT X))
                      (T (CLOSEF (OUTPUT X)))))
              (COND ((NULL A) T) (T A))))))

(DEFINEQ (MAP1
           (LAMBDA (FN L)
             (MAP L FN))))

(DEFINEQ (MAPC1
           (LAMBDA (FN L)
             (MAPC L FN))))

(DEFINEQ (MAPLIST1
           (LAMBDA (FN L)
             (MAPLIST L FN))))

(DEFINEQ (MAPCAR1
           (LAMBDA (FN L)
             (MAPCAR L FN))))

(DEFINEQ (DSKIN
           (NLAMBDA X
             (MAPC X (FUNCTION LOAD)))))

(DEFINEQ (ERRSET
           (NLAMBDA X
             (ERRORSET (CAR X)
                       (COND ((NULL (CDR X)) NIL)
                             (T (NULL (NULL (EVAL (CADR X))))))))))

(DEFINEQ (DEFPROP
           (NLAMBDA (I V P)
             (PROGN (PUT I P V) I))))

(DEFINEQ (TERPRI1
           (LAMBDA X
             (COND ((EQP X 0) (TERPRI))
                   (T (TERPRI) (ARG X 1))))))

(DEFINEQ (GREATERP1
           (LAMBDA (L)
             (OR (NULL L)
                 (NULL (CDR L))
                 (AND (GREATERP (CAR L) (CADR L))
                      (GREATERP1 (CDR L)))))))

(DEFINEQ (PROG3
           (LAMBDA X
             (ARG X 2))))

(DEFINEQ (DIVIDE
           (LAMBDA (X Y)
             (CONS (QUOTIENT X Y) (REMAINDER X Y)))))
```

```
(DEFINEQ (RECIP
           (LAMBDA (X)
             (QUOTIENT 1 X))))
(DEFINEQ (SIGN
           (LAMBDA (X)
             (COND ((EQP X 0) 0)
                   ((MINUSP X) -1)
                   (T 1)))))
(DEFINEQ (ROUND
           (LAMBDA (X)
             (TIMES (SIGN X) (FIX (PLUS (ABS X) 0.5))))))
(DEFINEQ (SUBST1
           (LAMBDA (X Y S)
             (COND ((EQUAL Y S) X)
                   ((ATOM1 S) S)
                   (T (CONS (SUBST1 X Y (CAR S)) (SUBST1 X Y (CDR S))))))))
(DEFINEQ (PRINT1
           (LAMBDA (X)
             (PROGN (TERPRI)
                    (PRIN2 X)
                    (PRIN1 (CHARACTER 32))))))
(DEFINEQ (TYO
           (LAMBDA (X)
             (PROGN (PRIN1 (CHARACTER X))
                    X))))
(DEFINEQ (ATOM1
           (LAMBDA (X)
             (OR (ATOM X) (STRINGP X)))))
(DEFINEQ (EQ1
           (LAMBDA (X Y)
             (OR (EQ X Y) (AND (FIXP X) (FIXP Y)) (EQP X Y)))))
(DEFINEQ (ERR
           (LAMBDA (X)
             (RETFROM (QUOTE ERRORSET) X))))
(DEFINEQ (CHRCT
           (LAMBDA NIL
             (DIFFERENCE (LINELENGTH) (POSITION (OUTPUT))))))
(DEFINEQ (EXPLODE
           (LAMBDA (X)
             (UNPACK X T))))
(DEFINEQ (FLATSIZE
           (LAMBDA (X)
             (NCHARS X T))))
(DEFINEQ (INTERN
           (LAMBDA (X)
             X)))
```

```
(DEFINEQ (REMOB
           (LAMBDA (X)
             X)))

(DEFINEQ (CSYM
           (NLAMBDA (X)
              (SETQ GENSYMCHAR (CAR (UNPACK X))))))

(DEFINEQ (GENSYM1
           (LAMBDA NIL
             (GENSYM GENSYMCHAR))))

(DEFINEQ (NCONS
           (LAMBDA (X)
             (CONS X NIL))))

(DEFINEQ (XCONS
           (LAMBDA (X Y)
             (CONS Y X))))

(DEFINEQ (PUTPROP
           (LAMBDA (X Y Z)
             (PUT X Z Y))))

(DEFINEQ (TYI
           (LAMBDA NIL
             (CHCON1 (READC)))))

(DEFINEQ (ZEROP1
           (LAMBDA (X)
             (EQP X 0))))

(DEFINEQ (READ1
           (LAMBDA NIL
             (READ NIL FILERDTBL))))

(DEFINEQ (GET1
           (LAMBDA (X Y)
             ((LAMBDA (A)
                (COND ((NULL A) A)
                      (T (CADR A))))
              (GETLIS X (NCONS Y))))))

(DEFINEQ (ASSOC1
           (LAMBDA (X Y)
             (COND ((NULL Y) NIL)
                   ((EQ1 X (CAAR Y))(CAR Y))
                   (T (ASSOC1 X (CDR Y)))))))

(DEFINEQ (SASSOC1
           (LAMBDA (X Y FN)
             ((LAMBDA (V)
                (COND ((NULL V) (FN))
                      (T V)))
              (ASSOC1 X Y)))))
```

REFERENCES

1. American Standards Association, *American Standard FORTRAN*, New York, 1966.
2. P. Naur (Ed.), 'Revised report on the algorithmic language ALGOL 60,' *Communications of the ACM*, May, 299–314 (1960).
3. J. McCarthy, 'Recursive functions of symbolic expressions and their computation by machine', *Communications of the ACM*, April, 184–195 (1960).
4. H. Samet, 'Proving the correctness of heuristically optimized code', *Communications of the ACM*, July, 570–581 (1978).
5. L. H. Quam and W. Diffie, *Stanford LISP 1.6 Manual*, Stanford Artificial Intelligence Project Operating Note 28.7, Computer Science Department, Stanford University, Stanford, California, 1972.
6. W. Teitelman, *INTERLISP Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California, 1978.
7. C. Wilcox, personal communication, 1976.
8. D. C. Smith, *MLISP*, Stanford Artificial Intelligence Project Memo AIM-135, Computer Science Department, Stanford University, Stanford, California, October 1970.
9. J. F. Reiser (Ed.), *SAIL*, Stanford Artificial Intelligence Project Memo AIM-289, Computer Science Department, Stanford University, Stanford, California, 1976.