APPENDIX B

THE LISP INTERPRETER

This appendix is written in mixed M-expressions and English. Its purpose is to describe as closely as possible the actual working of the interpreter and PROG feature. The functions evalquote, apply, eval, evlis, evcon, and the PROG feature are defined by using a language that follows the M-expression notation as closely as possible and contains some insertions in English.

```
evalquote[fn;args]=[get[fn;FEXPR]∨get[fn;FSUBR]→
                        eval[cons[fn;args];NIL]
                    T→apply[fn;args;NIL]]
```

This definition shows that evalquote is capable of handling special forms as a sort of exception. Apply cannot handle special forms and will give error A2 if given one as its first argument.

The following definition of apply is an enlargement of the one given in Section I. It shows how functional arguments bound by FUNARG are processed, and describes the way in which machine language subroutines are called.

In this description, spread can be regarded as a pseudo-function of one argument. This argument is a list. spread puts the individual items of this list into the AC, MQ, $ARG3, ... the standard cells for transmitting arguments to functions.

These M-expressions should not be taken too literally. In many cases, the actual program is a store and transfer where a recursion is suggested by these definitions.

```
apply[fn;args;a]=[
    null[fn]→NIL;
    atom[fn]→[get[fn;EXPR]→apply[expr[1];args;a];
                get[fn;SUBR]→ { spread[args];
                                $ALIST:=a;
                                TSX subr[1],4 } ;
                T→apply[cdr[sassoc[fn;a;λ[[];error[A2]]]];args;a];
    eq[car[fn];LABEL]→apply[caddr[fn];args;cons[cons[cadr[fn];caddr[fn]];a]];
    eq[car[fn];FUNARG]→apply[cadr[fn];args;caddr[fn]];
    eq[car[fn];LAMBDA]→eval[caddr[fn];nconc[pair[cadr[fn];args];a]];
    T→apply[eval[fn;a];args;a]]
```

---

1. The value of get is set aside. This is the meaning of the apparent free or undefined variable.

```
eval[form;a]=[
    null[form]→NIL;
    numberp[form]→form;
    atom[form]→[get[form;APVAL]→car[apval[1]];
               T→cdr[sassoc[form;a;λ[[ ];error[A8]]]]];
    eq[car[form];QUOTE]→cadr[form];[2]
    eq[car[form];FUNCTION]→list[FUNARG;cadr[form];a];[2]
    eq[car[form];COND]→evcon[cdr[form];a];
    eq[car[form];PROG]→prog[cdr[form];a];[2]
    atom[car[form]]→[get[car[form];EXPR]→apply[expr;[1] evlis[cdr[form];a];a];
                    get[car[form];FEXPR]→apply[fexpr;[1] list[cdr[form];a];a];
                    get[car[form];SUBR]→{spread[evlis[cdr[form];a]];
                                         $ALIST:=a;
                                         TSX subr,[1]4};
                    get[car[form];FSUBR]→{AC:=cdr[form];
                                          MQ:=$ALIST:=a;
                                          TSX fsubr,[1]4};
                    T→eval[cons[cdr[sassoc[car[form];a;λ[[];error[A9]]]];
                                cdr[form]];a]];
    T→apply[car[form];evlis[cdr[form];a];a]]
evcon[c;a]=[null[c]→error[A3];
    eval[caar[c];a]→eval[cadar[a];a];
    T→evcon[cdr[c];a]]
evlis[m;a]=maplist[m;λ[[j];eval[car[j];a]]]
```

The PROG Feature

The PROG feature is an FSUBR coded into the system. It can best be explained in English, although it is possible to define it by using M-expressions.

1. As soon as the PROG feature is entered, the list of program variables is used to make a new list in which each one is paired with NIL. This is then appended to the current a-list. Thus each program variable is set to NIL at the entrance to the program.

2. The remainder of the program is searched for atomic symbols that are understood to be location symbols. A go-list is formed in which each location symbol is paired with a pointer into the remainder of the program.

3. When a set or a setq is encountered, the name of the variable is located on the a-list. The value of the variable (or cdr of the pair) is actually replaced with the new value.

---

1. The value of get is set aside. This is the meaning of the apparent free or undefined variable.
2. In the actual system this is handled by an FSUBR rather than as the separate special case as shown here.

If the variable is bound several times on the a-list, only the first or most recent occurrence is changed. If the current binding of the variable is at a higher level than the entrance to the prog, then the change will remain in effect throughout the scope of that binding, and the old value will be lost.

If the variable does not occur on the a-list, then error diagnostic A4 or A5 will occur.

4. When a return is encountered at any point, its argument is evaluated and returned as the value of the most recent prog that has been entered.

5. The form go may be used only in two ways.

a. (GO X) may occur on the top level of the prog. x must be a location symbol of this prog and not another one on a higher or lower level.

b. This form may also occur as one of the value parts of a conditional expression, if this conditional expression occurs on the top level of the prog.

If a go is used incorrectly or refers to a nonexistent location, error diagnostic A6 will occur.

6. When the form cond occurs on the top level of a prog, it differs from other conds in the following ways.

a. It is the only instance in which a go can occur inside a cond.

b. If the cond runs out of clauses, error diagnostic A3 will not occur. Instead, the prog will continue with the next statement.

7. When a statement is executed, this has the following meaning, with the exception of the special forms cond, go, return, setq and the pseudo-function set, all of which are peculiar to prog.

The statement s is executed by performing eval[s;a], where a is the current a-list, and then ignoring the value.

8. If a prog runs out of statements, its value is NIL.

When a prog is compiled, it will have the same effect as when it is interpreted, although the method of execution is much different; for example, a go is always compiled as a transfer. The following points should be noted concerning declared variables.[1]

1. Program variables follow the same rules as λ variables do.

a. If a variable is purely local, it need not be declared.

b. Special variables can be used as free variables in compiled functions. They may be set at a lower level than that at which they are bound.

c. Common program variables maintain complete communication between compiled programs and the interpreter.

2. set as distinct from setq can only be used to set common variables.

---

1. See Appendix D for an explanation of variable declaration.