1984 note: the author of this is not JMC

THE COMPILER in LISP

1. The Compiler was written in LISP rather than SAP because the writer found it easier to make changes and additions particularly major ones, in LISP. In a program as large and complex as a compiler, manual simulation is extremely tedious, such changes were expected and one was not disappointed.

The scheme will ultimately be this:

(1) The present compiler is for the conventions of the May interpreter, but its subject matter will be S expressions written according to the latest conventions.

(2) This compiler will be translated by means of "inst" and "sublis" into the present conventions.

(3) Version A will compile version B using the May interpretator.

The output of the compiler is at present SAP.

2. Conventions

A number of changes have been made in conventions either because simpler SAP programs resulted or because they made it easier to write the compiler.

2.1 ARG2 is no longer in the MQ, but in a core location. The reason for this is that one cannot transfer directly between the MQ and index registers, and not even rapidly between the accumulator and MQ.

2.2 ARG1 is no longer the accumulator but Index Register 2. "car" and "cdr" then compile trivially. The null test does not require clearing the remainder of the accumulator. One can test whether a list in ARG1 is atomic without disturbing ARG1, and one can store IR4 on the PPDL without disturbing ARG1, and one can ~~store IR4 on the PPDL~~ test for EQ without disturbing ARG1.

2.3 Saving and unsaving are done by keeping the PPDL pointer permanently in IR1, and is compiled open.

2.4 Functional arguments are now represented by 15 bit quantities, which simplifies EQ, CONS, and CONSW since the prefix and address portions of ARG2 will always be clear. A functional argument will compile as

```
G0001, TXL, *+1,0,-*
              SXD, T1, 4
                    etc.
```

and will be applied to another argument by

```
LXD G0001,4
CLA 0,4
STA *+1
TSX **,4
```

2.5 "cons" is compiled open

3. Saving

The compiler must determine what registers need to be saved and preferably do this without look-ahead. The conventions adopted are therefore:

1) The registers in which the arguments are going to be stored are saved before such storage occurs, and restored before leaving the programs.

2) Self references or possible self references (unspecified functional arguments) are made transparent to intermediate results.

The handling of intermediate results is worth explaining. The need for temporary storage registers arises when one of the arguments of a form A requires the computation of another form B. Since the latter computation may use ARG1, ARG2, etc., any previously computed arguments of form A must be held temporarily elsewhere, and so on for any depth. But when the computation of a lower form is complete, the registers used by it become available for use by the higher form. Thus, by always using the first available temporary storage register and keeping a record of the last register used at the higher level one can arrange to save the minimum number of registers without the need for look-ahead.

Unsave and save instructions are generated by four small subroutines which can be replaced easily if a different convention is adopted.

4. Compiler Structure

4.1 The heart of the compiler consists of 3 functions, "~~fa~~unpile", "argpile" and "compile", which handle the compilation of functions, arguments and special forms respectively.

4.2 The bookkeeping is carried out by means of seven arguments A1 to A7. Before describing the purpose of each, a word on symbolic locations is appropriate. It was convenient to divide these into 3 classes

(1) Program locations, obtained from a symbol generator.

(2) Bound variable locations, obtained from a list of symbols of sufficient length (UL list)

(3) Intermediate results locations, obtained from a list of symbols of sufficient length. (TL list)

The latter two will eventually also be produced by the symbol generation.

4.3 The seven arguments.

A1: the LISP definition of the function, or part of the function, that is to be compiled

A2: The last symbol of the TL list used before compiling the current form. This information is requ$\overset{r}{i}$ed to determine how much must be saved if a self reference or a function which is an unspecified functional argument occurs.

A3: A list which is all or part of ARGL=($ARG2, $ARG3, etc.) which indicates the subargument that is currently being compiled.

A4: Is a list on which "move" instructions are assembled after the compilation of each subargument (with one exception) so that when subarguments have been compiled they will be moved back from TL locations to ARGL locations.

The exception is the last subargument which is moved immediately to ARGj.

A5: A list which is all or part of

TL = (T1, T2, T3, etc.)

which indicates the next available intermediate result storage location. (IRSL)

A6: Is a list of pairs, each pair consisting of a bound variable and the (UL) location in which the quantity that it represents is stored.

A7: Is a list of pairs, generated only by "LABEL", pairing the name of the function with the symbol generated for its beginning.

4.4 Funpile

If the function is "cons" then a sequence of nine instructions is inserted. If it is not atomic and begins with LAMBDA it is properly treated as a functional form. If the function is found on A6 then it is a functional argument and IRSLs are saved, the proper TSX instructions* is inserted and the IRSLs are unsaved. If the function is found on A7 a similar thing happens. If the location of the function has been predefined the proper TSX instruction is inserted; otherwise a symbol is generated for the function location which will be used in subsequent references to the same function and a SYN card is generated that defines the generated symbol as the name of the function.

The list of instruction on A4 is of course inserted in front of those generated by "funpile".

4.5 ARGPILE

If a subargument consists only of a previously bound variable then no instructions are generated, but the proper "move" is added to A4. Otherwise compilation takes place, the result is stored in car{A5} and instructions to move from car{A5} to car{A3} are added to A4. For this compilation A2 is set to car{A5}, and A4 is set to nil.

4.6 COMPILE

If the object program to be compiled is an atom then it may be OBJ, NIL or ERROR, which result in (, LXA, NILOBJ,2), (,LXD, NILOBJ,2) or ((,SXD,$ERROR,4),(,TSX,$ERROR,$),(,LXD,$ERROR,4)) respectively. Otherwise the object may be found on the list of bound variables or the list of predefined quantities. If not on these then a symbol is generated for its location which will always be used in subsequent references to this object and a SYN card is generated which defines the symbolic location as the name of the object. The instruction which is generated is always (,LXD,location,2)

If not atomic, then special forms are tested for. These are CAR, CDR, COND, QUOTE, LIST, ATOM, NULL, AND, OR, NOT, LAMBDA, LABEL, FUNCTION. If the object program is not atomic or a special form then it is assumed to be a function and "argpile" - "funpile" are executed.

In the bookkeeping of the compiler a good deal of use is made of the self restoring feature of lists. Specifically, bound variables and labels disappear from the arguments of "compile" when the functions in which they occur are left, the move instructions added to A4 are local and will not include move instructions ~~done~~ applicable to higher level forms, and the TL marker (A2) and A3 is restored whenever the compiler back-tracks to a higher level form.

5. <u>Conditionals</u>

Null and Atom are constructed under two circumstances. When COND is met, null, atom;eq and T generate the appropriate tests with transfer instructions to other object programs. However if any other predicate within COND occurs (beginning with COND, AND, OR, NOT) then truth or falsity is computed, using the ~~interest~~ internal representation ~~and~~ of -1 and zero for truth and falsity respectively, since these can be tested in the index registers using only one instruction.

As an example of a compilation the output for SUBST is attached. The LISP definition is

```
(LABEL, SUBST, (LAMBDA, (X, Y, Z), (COND, ((NULL, Z), NIL),
  ((ATOM, Z), (COND, ((EQ, Z, Y), X), (T, Z))),
  (T, (CONS, (SUBST, X, Y, (CAR, Z)), (SUBST, X, Y, (CDR, Z))))
                    )))
```

```
FUNCTION   APPLY(F,L,A) HAS BEEN ENTERED, ARGUMENTS..
EVAL
(((LAMBDA,(U,W),(PRINL1S,(APPEND,(CDR,(CL [illegible] ))),[illegible] ,(CONST,F),)),))),(ATTRIB,(CONST,F
N),(CONST,(SEQ,(F1,F2,F3,F4,F5,F6,F7,F8,F9,F10,F11,[illegible] ))),[illegible] ,(CONST,((LABEL,SUBST,(LAMBDA
,(X,Y,Z),(COND,((NULL,Z),IR2),((ATOM,IR2),(COND,((EQ,IR2,Y),X),[illegible] ,(CONS,(SUBST,X,Y,(CAR,IR2)),(SUBST,X,
Y,(CDR,Z)))))))))),(CONST,(LAMBDA,(V),(COMPILE,(CAR,V),[illegible] ,$ARG6,$ARG6,$ARG6,)))),((ARG0, ),(ARGL,($ARG1
,$ARG2,$ARG3,$ARG4,$ARG5,$ARG6)),(UL,(U1,U2,U3,U4,U5,U6,[illegible] ),[illegible] )),(QCA, )))
(( , ))

(F1,BSS,0)
(  ,PXD,0,4)
(  ,STO,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,LDQ,U1)
(  ,STQ,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,SXD,U1,2)
(  ,LDQ,U2)
(  ,STQ,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,LDQ,$ARG2)
(  ,STQ,U2)
(  ,LDQ,U3)
(  ,STQ,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,LDQ,$ARG3)
(  ,STQ,U3)
(  ,LXD,U3,2)
(  ,TXL,F3,2,0)
(  ,CLA,0,2)
(  ,COM)
(  ,ANA,NILOBJ)
(  ,TZE,*4)
(  ,CLA,0,2)
(  ,PAX,0,2)
(  ,SXD,$ARG3,2)
(  ,LDQ,U2)
(  ,STQ,$ARG2)
(  ,LXD,U1,2)
(  ,TSX,F1,4)
(  ,SXD,*1,2)
(  ,LXD,U3,2)
(  ,CLA,0,2)
(  ,PDX,0,2)
(  ,SXD,$ARG3,2)
(  ,LDQ,U2)
(  ,STQ,$ARG2)
(  ,LXD,U1,2)
(  ,LDQ,T1)
```

SAVE IR4

save U1

x

save U2

y [illegible]

[illegible]

[illegible]

(NULL, z) → NIL

NOTE THAT THE TEST FOR ATOM DOES NOT ALTER ARG1

(ATOM, z) →

(CAR, z)

$\overset{y}{x}$

subst[x; y; car[z]]

z

cdr[z]

y

x

[illegible]

[illegible]

[illegible]

```
FUNCTION  APPLY(F,L,A) HAS BEEN ENTERED, ARGUMENTS..
EVAL
(((LAMBDA,(U,A)),(PRINLIS,(APPEND,(CDR,(CDR,(CDR,(CONC,A)))),(APPEND,
N),(CONST,(SEQ,(F1,F2,F3,F4,F5,F6,F7,F8,F9,F10,F11,F12,F13,F14,F15,F1
,(X,Y,Z),(COND,((NULL,Z),IR2),((ATOM,IR2),(COND,((EQ,IR2,Y),X),(TRUE,I
1,(CDR,Z))))))))),(CONST,(LAMBDA,(V),(COMPILE,(CAR,V),$ARG1,$ARG2,$ARG3,
,$ARG2,$ARG3,$ARG4,$ARG5,$ARG6)),(U1,(U1,U2,U3,U4,U5,U6,U7,U8,U9)),(TL
(( , ))

(T1,BSS,0)
(  ,PXD,0,4)
(  ,STO,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,LDQ,U1)
(  ,STQ,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,SXD,U1,2)
(  ,LDQ,U2)
(  ,STQ,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,LDQ,$ARG2)
(  ,STQ,U2)
(  ,LDQ,U3)
(  ,STQ,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,LDQ,$ARG3)
(  ,STQ,U3)
(  ,LXD,U3,2)
(  ,TXL,F3,2,0)
(  ,CLA,0,2)
(  ,COM)
(  ,ANA,NILOCC)
(  ,TZE,F4)
(  ,CLA,0,2)
(  ,PAX,0,2)
(  ,SXD,$ARG3,2)
(  ,LDQ,U2)
(  ,STQ,$ARG2)
(  ,LXD,U1,2)
(  ,TSX,F1,4)
(  ,SXD,T1,2)
(  ,LXD,U3,2)
(  ,CLA,0,2)
(  ,PDX,0,2)
(  ,SXD,$ARG3,2)
(  ,LDQ,U2)
(  ,STQ,$ARG2)
(  ,LXD,U1,2)
(  ,LDQ,T1)
(  ,STQ,PPDL,1)
(  ,TNX,NOPDL,1,1)
(  ,TSX,F1,4)
(  ,TXI,*+1,1,1)
(  ,LDQ,PPDL,1)
(  ,STQ,T1)
(  ,SXD,$ARG2,2)
(  ,LXD,T1,2)
(  ,PXD,0,2)
```

} SAVE IR4

save U1

z

save U2

y to U2

~~save~~ U3

z to U3

(NULL, z) → NIL

} NOTE THAT THE TEST FOR ATOM DOES NOT ALTER ARG1

(ATOM, z) →

} (CAR, z)

x y

} subst[x; y; car[z]]

z

cdr[z]

y

x

} before computing "subst" again, save the value of the previous "su

subst[x; y; cdr[z]]

} and restore T1

ARG2 = subst[x; y; cdr[z]]
ARG1 = subst[x; y; car[z]]

```
(  ,ARS,18)
(  ,ADD,$ARG2)
(  ,LXD,$FREE,2)                          the open cons[  ;  ]
(  ,TXH,*+2,2,0)
(  ,TSX,$FROUT,4)
(  ,LDQ,0,2)
(  ,STQ,$FREE)
(  ,STO,0,2)
(F2,BSS,0)
(  ,TXI,*+1,1,1)
(  ,LDQ,PPDL,1)
(  ,STQ,U3)
(  ,TXI,*+1,1,1)
(  ,LDQ,PPDL,1)                              RESTORE U1, U2, U3 as they were before
(  ,STQ,U2)                                     the routine was entered
(  ,TXI,*+1,1,1)
(  ,LDQ,PPDL,1)
(  ,STQ,U1)
(  ,TXI,*+1,1,1)                        UNSAVE ,R4
(  ,CLA,PPDL,1)
(  ,PDX,0,4)
(  ,TRA,1,4)                               RETURN
(F3,BSS,0)
(  ,TRA,F2)                             (NULL, z) → NIL , go to restore and unsave
(F6,BSS,0)
(  ,LXD,U1,2)
(  ,TRA,F5)
(F4,BSS,0)                              (ATOM, z) →
(  ,SXD,T1,2)
(  ,LXD,U2,2)
(  ,SXD,$ARG2,2)
(  ,LXD,T1,2)
(  ,PXD,0,2)
(  ,SUB,$ARG2)
(  ,TZE,F6)                             z = y → x
(F5,BSS,0)
(  ,TRA,F2)                             T → z    SINCE z was still in ARG1

  END OF APPLY, VALUE IS ...
```