A simple list-processing interpreter

By D. M. Foster*

Much of the Extended Mercury Autocode Compiler for Orion is written in a list-processing language. This paper describes the language and the Interpreter that was written to interpret it.

The Interpreter was designed to facilitate the writing of the Extended Mercury Autocode Compiler for Orion. The central core of the EMA Compiler builds arithmetic expressions into list-structure trees and then breaks them down into machine instructions. This building up and breaking down is performed by functions written in a list-processing language similar to "LISP" (McCarthy, 1960; Woodward and Jenkins, 1961). The Interpreter was written to interpret this language.

Furthermore, the compiler was originally required to work in an Orion with a 4K core store. In order to make the best use of the store available the original list-processing language had to be translated into as compact a form as possible and the Interpreter had to work on functions in this form.

Lists

These are strings of items written in the form \((a, b, c, d, \ldots, x)\) where \(a, b, c, \ldots\) may themselves be lists or individual items called atoms. In the Orion EMA Compiler a list is stored in the machine as a number of words, where there is one word for each item of the list. The least-significant half of such a word contains the address of the next word in the list, and in the case of the last word the least significant half is zero or \(NIL\).

The most-significant half of each word represents the item itself. If the item is an atom, the bit pattern of the half-word represents that atom. If the item is a list, the half-word is the address of the first word of that list.

For example consider the list

\[(A, (B, C), (D, (E, F)))\]

where \(A, B, C, D, E, F\) are atoms. It is a list of three items. The first item is the atom \(A\). The second item is a list itself consisting of two items, the atoms \(B\) and \(C\). The third item is a list of two items, the atom \(D\) and a list of two items, the atoms \(E\) and \(F\). This list would be represented in the computer as in Fig. 1. The arrows point from half-words containing addresses to the words addressed.

An example of the type of list produced by the EMA compiler is

\[(G, (+_{G}, (C, x^*), (/_{G}, (C, y^*), (C, (+_{I}, a^*_0, (C, i^*))))))\]

which is represented in the computer as in Fig. 2. This list would result from the arithmetic expression

\[x + y/ai\]

where \(i\) is a subscript. \(x^*, y^*, a^*_0\) and \(i^*\) give the addresses of the locations of the EMA quantities \(x, y, a_0\) and \(i\) in the store. \(C\) is an atom denoting "Contents of" which always occupies the first position in a 2-item list. The second item of such a list is then taken to be an address, and the list represents the contents of that address. \(+_{G}\) and \(+_{I}\) are operators signifying floating-point addition, floating-point division and integer addition, respectively. They occupy the first position in 3-item lists and denote operation on the other two items. \(G\) occupies the first position in a 2-item list, and indicates that the expression represented by the second item is a floating-point expression.

The first item of a list is known as its operator and the subsequent items as its operands. The reason for this convention is demonstrated by the above example.

The interpreted functions

The lists are manipulated by functions that can act upon them. The arguments of these functions are lists or atoms. The result of a function is, in general, a single list or atom.

Most of the functions are defined in terms of other functions. For example, we might define the function \texttt{fn1} by writing

\[
\texttt{fn1}[x;y] = (\texttt{fn2}[x], \texttt{fn3}[y])
\]

i.e. \texttt{fn1} is a function with two arguments \texttt{x} and \texttt{y} and is defined to be a list of two items, each of which is the result of another function acting on one of the arguments of \texttt{fn1}. In the notation used here, square brackets are used to enclose the arguments of functions.

The action of some functions is not defined in this way but is implicit in the language. Such functions are called \textit{machine functions}. \texttt{Optr, 1st op} and \texttt{rem} are examples of machine functions. \texttt{Optr[x]} gives the operator of the list \texttt{x}, i.e. its first item. \texttt{1st op[x]} gives the first operand of the list, i.e. its second item. \texttt{Rem[x]} gives the list that remains after the first item has been removed from the list \texttt{x}. E.g. if \texttt{x} = (\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}), then \texttt{optr[x]} = \texttt{A}, \texttt{1st op[x]} = \texttt{B}, \texttt{C} and \texttt{rem[x]} = (\texttt{B}, \texttt{C}, \texttt{D}).

When a list \texttt{x} consists of only two items \texttt{1st op[x]} is by convention simply written as \texttt{op[x]}. If \texttt{x} is a list of only one item, the result of \texttt{rem[x]} will be the atom \texttt{NIL}.

Using these functions, further functions can be defined. E.g.

\[
\texttt{2nd op[x] = 1st op[rem[x]]}
\]
defines \texttt{2nd op} which gives the second operand of a list.

There are also Boolean functions whose answer is one of the two atoms \texttt{True} or \texttt{False}. E.g. the machine function \texttt{atom[x]} gives the result \texttt{True} if \texttt{x} is an atom and the result \texttt{False} otherwise. Thus a function may be written.

\[
\texttt{fn 4 [x; y] = atom [x]} \rightarrow \texttt{fn 5 [x; y]}
\]

\[
T \rightarrow \texttt{fn 5 [optr [x] ; y]}
\]
i.e. if \texttt{x} is an atom the result of the function is the first alternative, \texttt{fn 5 [x; y]}, and otherwise the result is the second alternative, \texttt{fn 5 [optr [x] ; y]}. Thus since (\texttt{A}, \texttt{B}) is not an atom,

\[
\texttt{fn 4 [(A, B); C] = fn 5 [optr [(A, B)]; C]}
\]

\[
= \texttt{fn 5 [(A); C]}
\]

\textit{T} represents the atom \texttt{True}.

\texttt{Equals[x; y]} is a second Boolean machine function which gives the answer \texttt{True} if \texttt{x} and \texttt{y} are equal and \texttt{False} otherwise. The answer \texttt{True} may indicate that \texttt{x} and \texttt{y} are the same atom or that they are identical lists.

There are several other common Boolean machine functions. \texttt{And[x; y]} gives the answer \texttt{True} if and only if \texttt{x} and \texttt{y} are both \texttt{True}. \texttt{Or[x; y]} gives the answer \texttt{True} if either \texttt{x} or \texttt{y} are \texttt{True}. \texttt{Eq2[x; a; b]} is \texttt{True} if \texttt{x} is equal to either \texttt{a} or \texttt{b}. \texttt{Eq3[x; a; b; c; \ldots; t]} is \texttt{True} if \texttt{x} is equal to one of \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{t}.

If \texttt{x} is an atom the result of both \texttt{optr[x]} and of \texttt{rem[x]} is a special atom \texttt{Rubatoma}, meaning a rubbish atom.

\texttt{1st op[x]} will also give the answer \texttt{Rubatoma} if \texttt{x} is an atom or a list of one item. This enables us to write, for example,

\texttt{Equals[G; optr[x]]}

even if we do not know that \texttt{x} is a list with an obtainable first item, and to obtain the result \texttt{True} only if there is such an item and it is equal to \texttt{G}.

The language is recursive, i.e. functions may be defined in terms of themselves. E.g. we may define \texttt{fn 7} as

\[
\texttt{fn 7[x] = equals [rem[x]; NIL]} \rightarrow x
\]

\[
T \rightarrow \texttt{fn 7 [rem[x]]}
\]

The result of this function is the final item of the list \texttt{x}.

Some examples will show how these functions work. First, however, a set of functions that will be used in the examples is given. This set of functions is not, in fact, a subset of the actual functions used in the EMA Compiler, but comprises a simplified set that will demonstrate the sort of way in which the Compiler works. These functions will deal with a restricted form of EMA arithmetic instruction that consists only of simple arithmetic operations on integer and floating-point quantities, and that does not contain subscripts. Also these functions are restricted to those that break down into Orion machine orders the trees that represent arithmetic expressions. For reasons of space, the way in which such trees are built up is not discussed.

\textbf{A set of functions}

\textbf{Assign[x; y] =}

\[
\texttt{and [equals [optr [y] ; I]; equals [optr [op [y]] ; C]]}
\]

\[
\rightarrow \texttt{copy [I-form [x] ; op [op [y]] ; S0]} \]

\[
\texttt{and [equals [optr [y] ; G]; equals [optr [op [y]] ; C]]}
\]

\[
\rightarrow \texttt{copy [G-form [x] ; op [op [y]] ; S0]} \]

\[
T \rightarrow \texttt{note-error [1]}
\]

\textbf{I-form [x] =}

\[
\texttt{equals [optr [x] ; I]} \rightarrow \texttt{op [x]}
\]

\[
\texttt{equals [optr [x] ; G]} \rightarrow (\texttt{Int, op [x]})
\]

\[
T \rightarrow \texttt{note-error [2]}
\]

\textbf{G-form [x] =}

\[
\texttt{equals [optr [x] ; I]} \rightarrow (\texttt{Float, op [x]})
\]

\[
\texttt{equals [optr [x] ; G]} \rightarrow \texttt{op [x]}
\]

\[
T \rightarrow \texttt{note-error [3]}
\]

\textbf{Copy[x; y; S] =}

\[
\texttt{atom [x]} \rightarrow \texttt{accept 2 [14; y; x]}
\]

\[
\texttt{equals [optr [x] ; C]} \rightarrow \texttt{accept 2 [04; y; op [x]]}
\]

\[
\texttt{equals [optr [x] ; Int]} \rightarrow \texttt{instr B [103; store [op [x] ; S] ; 147; y; S]}
\]

\[
\texttt{equals [optr [x] ; Float]} \rightarrow \texttt{instr B [102; store [op [x] ; S] ; 47; y; S]}
\]

\[
\texttt{eq4 [optr [x] ; a; b]} \rightarrow \texttt{instr B [opco [optr [x]] ; store [1st op [x] ; S] ; 2nd op [x] ; S ; y; S]}
\]

\[
\texttt{eq3 [optr [x] ; a; b]} \rightarrow \texttt{instr A [opco [optr [x]] ; 1st op [x] ; 2nd op [x] ; y; S]}
\]

\[
T \rightarrow \texttt{note-error [4]}
\]
List-processing interpreter

\[
\text{Store } [x; S] = \\
\quad \text{atom } [x] \rightarrow \text{allocate } [x] \\
\quad \text{equals } [\text{optr } [x] ; C] \rightarrow \text{op } [x] \\
\quad T \rightarrow \text{store } A \begin{array}{l} \text{same } [S] ; \text{next } [S] ; x \end{array}
\]

\[
\text{Instr } A [f; x; y; z; S] = \\
\quad \text{atom } [y] \rightarrow \text{instr } B \begin{array}{l} \text{one } [f] ; \text{store } [x; S] ; y ; z ; S \end{array} \\
\quad \text{atom } [x] \rightarrow \text{instr } B \begin{array}{l} \text{twist } [f] \end{array} ; \text{store } [y; S] ; \text{next } [S] ; \text{next } [S] ; x ; z ; S \\
\quad T \rightarrow \text{instr } B \begin{array}{l} f ; \text{store } [x; S] ; \text{store } [y; S] ; z ; S \end{array}
\]

\[
\text{Instr } B [f; x; y; z; S] = \\
\quad \text{equals } [x; z] \rightarrow \text{accept } 2 \begin{array}{l} [f; x; y] \end{array} \\
\quad \text{and } [\text{eq6 } [f; 00; 01; 30; 90; 91; 94]; \text{equals } [y; z]] \\
\quad \quad \rightarrow \text{accept } 2 \begin{array}{l} \text{twist } [f] ; y ; x \end{array} \\
\quad \text{short } [z] \rightarrow \text{accept } 3 \begin{array}{l} [f; x; y; z] \end{array} \\
\quad T \rightarrow \text{accept } 2 \begin{array}{l} \text{next } [S] ; \text{result } A \begin{array}{l} \text{same } [S] \end{array} \end{array}
\]

\[
\text{Result } A [S; f; x; y] = \\
\quad \text{result } \text{instr } B \begin{array}{l} [f; x; y; z; S] \end{array}
\]

\[
\text{Opco } [x] = \text{equals } [x; +] \rightarrow 90 \\
\quad \text{equals } [x; -] \rightarrow 91 \\
\quad \text{equals } [x; \times] \rightarrow 94 \\
\quad \text{equals } [x; /] \rightarrow 95 \\
\quad \text{equals } [x; +] \rightarrow 00 \\
\quad \text{equals } [x; -] \rightarrow 01 \\
\quad T \rightarrow 30
\]

\[
\text{One } [x] = \text{equals } [x; 00] \rightarrow 10 \\
\quad \text{equals } [x; 01] \rightarrow 11 \\
\quad \text{equals } [x; 02] \rightarrow 12 \\
\quad T \rightarrow 34
\]

\[
\text{Twist } [f] = \text{equals } [x; 01] \rightarrow 02 \\
\quad \text{equals } [x; 91] \rightarrow 92 \\
\quad T \rightarrow f
\]

Some examples

(i) Consider the EMA instruction

\[
a = b - a
\]

where \(a\) and \(b\) are both floating-point quantities. The right-hand side will be built up into the following tree

\[
(G, (-o, (C, b*), (C, a*)))
\]

and the left-hand side into

\[
(G, (C, a*))
\]

The function assign is used to break down pairs of trees such as this into machine instructions. The trees representing the right-hand side and the left-hand side become the first and second operands, respectively, of assign. The Interpreter has to evaluate

\[
\text{assign } (G, (-o, (C, b*), (C, a*))); (G, (C, a*)) \quad (1)
\]

Now \(\text{optr } [(G, (C, a*))] = G\) and \(\text{optr } [\text{optr } ((G, (C, a*))] = C\), and so the second set of conditions in the definition of assign is satisfied. Therefore, using this definition, (1) becomes

\[
\text{copy } [(G, (-o, (C, b*), (C, a*))); S_0] \\
\text{op } [(G, (C, a*))]; S_0 \quad (2)
\]

Here \(S_0\) is an atom indicating a series of accumulators that are available for intermediate results when arithmetic expressions are being evaluated.

Looking at the definition of G-form it can be seen that

\[
\text{G-form } [(G, (-o, (C, b*), (C, a*))); S_0] \\
\text{op } [(G, (C, a*))]; S_0
\]

and so (2) reduces to

\[
\text{copy } [(G, (-o, (C, b*), (C, a*))); a*]; S_0 \quad (3)
\]

Now \(\text{eq4 } [\text{optr } [(G, (C, b*), (C, a*))]; +, -; \times, \\
S_0] = \text{True}
\]

and so the 5th possibility in the definition of copy is taken and (3) becomes

\[
\text{instr } B \begin{array}{l} \text{opco } \text{optr } [(G, (C, b*), (C, a*))]; \text{store } [1st \text{op } \begin{array}{l} [(G, (C, b*), (C, a*)]); S_0] \end{array} \end{array} \end{array} \\
\text{op } [(G, (C, a*))]; S_0 \quad (4)
\]

The effect of opco is to give the appropriate instruction code for an Orion machine order. 91 is the code for floating-point subtraction.

Orion machine instructions can either take a 3-address form, or a 2-address form. In the first case the result of an operation on the first two operands is written into a third address. This third address occupies only 6 bits, and so it is restricted to one of the first 64 registers in the store the which are called accumulators.

In 2-address instructions the result of the operation overwrites the first operand. In this case the addresses may be modified by the contents of the third address. E.g. the 3-address instruction

\[
91 \text{ A100 A200 A1}
\]

subtracts the contents of the register A200 from the contents of A100 and places the result in A1. The unmodified 2-address instruction

\[
91 \text{ A100 A200}
\]

performs the same operation but places the result in A100.

The function instr B first tests an embryonic instruction to see if the destination is the same as one of the operands and so a 2-address form is possible. Otherwise it tests to see if the destination is an accumulator and a 3-address form is possible. If neither of these conditions is satisfied, the operation will have to be broken down into two instructions.
List-processing interpreter

Sometimes the destination is found to be the same as the second operand, and in some cases it is then possible to interchange the first and second operands to enable a 2-address instruction to be formed. It may then be necessary to alter the instruction code, and the function twist will do this if such an alteration is required. This is what happens in the case of the present example, where the instruction code 91, which subtracts the second operand from the first, is replaced by the code 92, which subtracts the first from the second.

Now the second set of conditions in the definition of instr B is satisfied by (4) since both 91 is one of the six specified codes and the third and fourth arguments of the function are equal. So (4) becomes

\[ \text{accept 2 [twist [91] ; a*; b*]} \]

which reduces to

\[ \text{accept 2 [92; a*; b*]} \]

Accept 2 is the function that outputs a compiled unmodified 2-address Orion machine instruction. The original EMA instruction has been compiled into a single machine instruction that subtracts the contents of a* from the contents of b* and leaves the answer in a*.

(ii) Consider the EMA instruction

\[ i = j + 3k \]

where \( i, j \) and \( k \) are all integer quantities.

This gives rise to

\[ \text{assign [(I, (+j, (C, j*)), (×i, 3, (C, k*))); (I, (C, i*)]}} \]

which becomes

\[ \text{copy [I-form [(I, (+j, (C, j*)), (×i, 3, (C, k*))); (I, (C, i*)]]} \]

\[ \text{op [op [(I, (C, i*)]); S0]} \]

\[ = \text{copy [(+j, (C, j*)), (×i, 3, (C, k*)); S0]} \]

\[ = \text{instr A [opco [optr [(+j, (C, j*)), (×i, 3, (C, k*)); S0], 1st op [+(j, (C, j*)), (×i, 3, (C, k*)); S0], 2nd op [(+(j, (C, j*)), (×i, 3, (C, k*)); S0]]]; S0] \]

\[ = \text{instr B [00; (C, j*); (×i, 3, (C, k*)); S0]}} \]

The function instr A tests an embryonic machine instruction to see if one of its operands is an atom so that an instruction can be formed that acts directly on a number rather than on the contents of an address. If this is possible the instruction code has to be changed and this is done by the function one. Also the atom has to be the second operand of the instruction, and so if the first operand is an atom, the first and second operands have to be interchanged and the function twist used. In this case no atoms are found and (6) becomes

\[ \text{instr B [00; store [(C, j*); S0]; store [(×i, 3, (C, k*)); S0; S0]} \]

\[ = \text{instr B [00; j*; store A [same [S0]; next [S0]; (×i, 3, (C, k*)); i*; S0]}} \]

The result of the machine function next [S] is an atom representing the next item of the series S. The atom given by next [S0] is denoted by s0 which represents the first accumulator available for intermediate results. The residual series becomes S1. The result of same [S] is to construct another copy of the series S that will not be affected by the operation of next on the original series. Hence (7) becomes

\[ \text{instr B [00; j*; store A [S0; s0; (×i, 3, (C, k*)); i*; S1]} \]

\[ = \text{instr B [00; j*; result [copy [(×i, 3, (C, k*)); s0; S0]}; s0]; i*; S1} \]

(8)

Result is a special function with two arguments. It causes the evaluation of its first argument and gives its second argument as an answer. It has the effect of splitting (8) into the two functions

\[ \text{copy [(×i, 3, (C, k*)); s0; S0]}} \]

which is evaluated first, and

\[ \text{instr B [00; j*; s0; i*; S1]}} \]

(9) becomes

\[ \text{instr A [opco [×i]; 3; (C, k*)]; s0; S0] \]

\[ = \text{instr A [30; 3; (C, k*); s0; S0]}} \]

and since 3 is an atom this gives

\[ \text{instr B [one [twist [30]]; store [(C, k*); S0]; 3; s0; S0] \]

\[ = \text{instr B [34; k*; 3; s0; S0] \]

\[ = \text{accept 3 [34; k*; 3; s0] \]

since short [s0] is True as s0 is an accumulator. Since i* is also an accumulator short [i*] is True and (10) becomes

\[ \text{accept 3 [00; j*; s0; i*] \]

Hence the EMA instruction (5) is compiled into two 3-address Orion machine instructions. The first multiplies k by 3 and leaves the result in an accumulator. The second places j plus this result in i*.

(iii) Take the EMA instruction

\[ x = i \]

where \( x \) and \( i \) are floating-point and integer quantities, respectively. In this case we have to evaluate

\[ \text{assign [(I, (C, i*)); (G, (C, x*))] \]

\[ = \text{copy [G-form [(I, (C, i*)); op [op [(G, (C, x*)); S0] \]

\[ = \text{copy [float, op [(I, (C, i*)); x*; S0] \]

\[ = \text{copy [float, (C, i*)); x*; S0] \]

\[ = \text{instr B [102; store [(C, i*); S0]; 47; x*; S0] \]

\[ = \text{instr B [102; i*; 47; x*; S0] \]

\[ = \text{accept 2 [04; x*; result A [S0; s0; 102; i*; 47] \]

\[ = \text{accept 2 [04; x*; result [instr B [102; i*; 47; s0; S0]; s0] \]

\[ = \text{instr B [102; i*; 47; s0; S0] \]

\[ = \text{accept 2 [04; x*; s0] \]

\[ = \text{accept 3 [102; i*; 47; s0; s0] \]

\[ = \text{accept 2 [04; x*; s0] \]

\( x* \) is not an accumulator and so cannot occupy the third address position of a 3-address machine instruction. Hence two instructions are compiled. The 102 instruction floats i and puts the result in s0, and the 04 instruction transfers the contents of s0 to x*.
(iv) As a final example consider the EMA instruction
\[ a = a + (4 - b)(3 - i) \]

Here \( a \) and \( b \) are floating-point quantities and \( i \) is an integer quantity. When the tree that represents the right-hand side of the instruction is formed, a register, say \( c_i \), in a table of constants is allocated to hold 4 in floating-point form. We have to evaluate.

\[
\text{assign} \quad [(G, (+, 0), (C, a^*)), (G, (-, 0), (C, c_i), (C, b^*)),
\quad (\text{Float}, (r, 3, (C, i^*)))); (G, (C, a^*))]
\]

\[
\text{copy} \quad [((+, 0), (C, a^*)), (G, (-, 0), (C, c_i), (C, b^*))],
\quad (\text{Float}, (r, 3, (C, i^*);
\quad a^*, s_0)]
\]

\[
\text{instr } B \text{ [opco] } [+ \text{ o }]; \text{ store } [(C, a^*); s_0];
\quad \text{store} \quad [(G, (-, 0), (C, c_i), (C, b^*)),
\quad (\text{Float}, (r, 3, (C, i^*)); a^*, s_0); a^*, s_0]
\]

\[
\text{instr } B \quad [90]; a^*; \text{ store } A \quad [s_0; s_0];
\quad \text{store} \quad [(G, (-, 0), (C, c_i), (C, b^*)),
\quad (\text{Float}, (r, 3, (C, i^*)); s_0; s_0); a^*, s_1]
\]

\[
\text{instr } B \quad [90]; a^*; \text{ result } \text{ copy } [((G, (-, 0), (C, c_i), (C, b^*)),
\quad (\text{Float}, (r, 3, (C, i^*))); s_0; s_0); a^*, s_1]
\]

\[
\text{copy} \quad [(G, (-, 0), (C, c_i), (C, b^*)); s_0; s_0]
\]

\[
\text{result } \text{ copy } [((\text{Float}, (r, 3, (C, i^*)); s_1; s_1)]
\]

\[
\text{instr } B \quad [95]; s_0; s_1; s_0; s_2]
\]

The second of this pair of functions is now fully evaluated. The first becomes

\[
\text{instr } B \quad [95]; \text{ store } A \quad [s_0; s_0; (G, (-, 0), (C, c_i), (C, b^*));
\quad \text{store} \quad [(S_1; s_1; (\text{Float}, (r, 3, (C, i^*)); s_0; s_0)]
\]

\[
\text{ instr } B \quad [95]; \text{ result } \text{ copy } [((G, (-, 0), (C, c_i), (C, b^*)); s_0; s_0)]
\]

\[
\text{copy} \quad [(G, (-, 0), (C, c_i), (C, b^*)); s_0; s_0]
\]

\[
\text{copy} \quad [((\text{Float}, (r, 3, (C, i^*)); s_1; s_1)]
\]

\[
\text{instr } B \quad [95]; s_0; s_1; s_0; s_2]
\]

The first of these functions becomes

\[
\text{instr } B \quad [\text{opco} - \text{ o }]; \text{ store } [(C, c_i); s_0]; \text{ store } [(C, b^*);
\quad (\text{Float}, (r, 3, (C, i^*)); s_0; s_0)]
\]

\[
\text{instr } B \quad [91]; c_i; b^*; s_0; s_0]
\]

\[
\text{accept } 3 \quad [91]; c_i; b^*; s_0; s_0]
\]

The second function of (12) becomes

\[
\text{instr } B \quad [102]; \text{ store } [(\text{Float}, (r, 3, (C, i^*)); s_1); 47; s_1; s_1]
\]

\[
\text{instr } B \quad [102]; \text{ store } A \quad [s_1; s_1; (\text{Float}, (r, 3, (C, i^*));
\quad 47; s_1; s_1)]
\]

\[
\text{instr } B \quad [102]; \text{ result } \text{ copy } [((\text{Float}, (r, 3, (C, i^*)); s_1; s_1)]
\]

\[
\text{copy} \quad [(\text{Float}, (r, 3, (C, i^*)); s_1; s_1)]
\]

\[
\text{instr } B \quad [102]; s_1; 47; s_1; s_1]
\]

\[
\text{instr } A \quad [\text{opco} - \text{ o }]; 3; (C, i^*); s_1; s_1]
\]

\[
\text{accept } 2 \quad [95; s_0; s_1]
\]

\[
\text{accept } 3 \quad [91; c_i; b^*; s_0]
\]

\[
\text{accept } 3 \quad [12; i^*; 3; s_1]
\]

\[
\text{accept } 2 \quad [102; s_1; 47]
\]

\[
\text{accept } 2 \quad [95; s_0; s_1]
\]

\[
\text{accept } 2 \quad [90; a^*; s_0]
\]

I.e. a first instruction is compiled to subtract the contents of \( b^* \) from the contents of \( c_i \) and to leave the answer in \( s_0 \). The second instruction subtracts the contents of \( i^* \) from 3 and leaves the answer in \( s_1 \). The third instruction floats the contents of \( s_1 \). The fourth divides the contents of \( s_0 \) by the contents of \( s_1 \). The fifth adds the contents of \( s_0 \) to the contents of \( a^* \). The first, fourth and fifth of these instructions perform floating-point operations. The second instruction performs a fixed-point operation.

The working and free stores

The Interpreter handles the functions using two blocks of store which we call the Working Store and the Free Store. The Working Store is a push-down list. It is filled up consecutively, word by word, and at any time the last item that has been placed on the Working Store is called the top word of the store. Action always takes place on the top few words of the Working Store and the words lower down are not touched until the top words have been removed. A certain register contains what is known as the Working Store Pointer, which is the address of the top word of the store. The top word of the Working Store is also known as the first word of the store, the next word is known as the second word and so on.

A function expects to find its arguments as the first few words of the Working Store, and it replaces them with its result. Normally a function has one result. Hence a function with \( n \) arguments would have the effect of reducing the size of the Working Store by \( n - 1 \). In the case when a function has no arguments, the size of the Working Store is increased by one. E.g. consider the function \[ s \] that was defined above. When it is called in its arguments \( x \) and \( s \) will occupy the second and first words of the Working Store, respectively: i.e. the Working Store Pointer points to the word containing \( s \). On exit from the function its answer will have overwritten \( x \) and the Working Store Pointer will have been reduced by one so that it points to this answer.

The Free Store is used to supply extra words when these are needed to build up lists. For example there is a function that operates on the EMA identifier \( a \) to produce the list \( G_i(C, a^*) \). This is represented in the
machine as in Fig. 3, and so the function will require 4 words from the Free Store to build it up. The function will then replace the atom $a$ on the Working Store by an address pointing to this structure of words in the Free Store.

If an identifier comes twice in an arithmetic instruction, the second time it appears a new list will not be created, but the word that is placed on the Working Store to represent it will point to the list that is already in the Free Store. This treatment of identical lists not only saves space but also enables identical lists to be recognized by the machine function equals, since the words on the Working Store that represent the lists will themselves be identical. By this means common sub-expressions are recognized in certain cases.

For example consider the instruction

$$a(i + j) = a(i + j) + b(i + j)$$

where $(i + j)$ is a subscript. The left- and right-hand sides are, respectively, built up into the lists

$$(G, (C, (+_j, a_0^*, (+_j, (C, i^*), (C, j^*)))), (C, (+_j, b_0^*, (+_j, (C, i^*), (C, j^*))))$$

and

$$(G, (C, (+_j, a_0^*, (+_j, (C, i^*), (C, j^*)))), (C, (+_j, b_0^*, (+_j, (C, i^*), (C, j^*))))$$

These lists will be represented in the Free Store as in Fig. 4. The words in the Working Store that represent the left- and right-hand sides will point to the words at (i) and (ii) respectively. When these lists are broken down the destination is recognized as being the same as the first operand, and the subscript $(i + j)$ is recognized as a common sub-expression. The instruction is compiled into two Orion machine instructions. The first adds $i$ and $j$ and leaves the results in an accumulator. The second is a 2-address modified instruction where both addresses are modified by the accumulator containing $i + j$.

A Free Store Pointer indicates the next Free Store word available. This is reset to its initial value after the evaluation of every line of EMA program.

The stored form of the functions

The functions are stored in a compact form as a sequence of packed 6-bit and 12-bit instructions which are obeyed sequentially by the Interpreter. For example, a certain 6-bit instruction indicates, “Call in the machine function optf to act on the top item of the Working Store.” This instruction would result in the top item $x$ of the Working Store being replaced by $optf[x]$.

These instructions can be denoted by the form $L_n$, where $L$ is a letter and $n$ an integer. We denote the instructions that call in machine functions by $M_n$, where
the non-negative integer \( n \) specifies which machine function is required. Other types of instruction are denoted by other letters. A list of the different possible types of instruction follows.

\( F_n \) Call in the function specified by the integer \( n \) to act on arguments at the top of the Working Store.

\( M_n \) Call in the machine function specified by the integer \( n \) to act on arguments at the top of the Working Store.

\( X_n \) Take the \((k-n)\)th argument of the current function and place it on the top of the Working Store. \( k \) is the number of arguments of the function plus one; e.g. in the function \( \text{Copy}[x;y;S] \) the instruction \( X_1 \) would mean, “Place \( S \) on the top of the Working Store.” Similarly \( X_2 \) would refer to \( y \) and \( X_3 \) to \( x \).

This order of the numbering is the reverse of what might be expected. However, it is in fact the natural order. When a function is called in its arguments have been placed on the Working Store one by one and the last argument is at the top. So \( X_1 \) refers to the argument that was the first word of the Working Store at the time the function was called in, \( X_2 \) refers to the argument that was the second word, and so on.

\( C_n \) Take the constant specified by the integer \( n \) and place it on the top of the Working Store. This constant will be an atom such as, for example, the atom representing \( +1 \).

\( D_n \) Place the atom representing the decimal integer given by \( n \) on the Working Store.

\( J_n \) This is a conditional jump instruction. Jump to the instruction \( n \) 6-bit characters on unless the top item of the Working Store is the atom \( \text{True} \). Reduce the Working Store Pointer by one.

\( A_n \) Replace the \((n+1)\)th item on the Working Store by the atom \( \text{True} \) if this item is identical with one of the first \( n \) items. Otherwise replace it by the atom \( \text{False} \). Adjust the Working Store Pointer to point to this atom.

\( E_n \) Terminate the current function and place its result on the Working Store, overwriting the first of its arguments. The Working Store Pointer will then point to this result and control will be returned to the higher-level function. \( n \) is the number of operands of the function to be terminated.

The instructions \( X_n \) where \( n < 5 \), \( E_n \) where \( 1 \leq n < 5 \), and \( M_n \) where \( n \leq 63 \) are 6-bit instructions. All the others are 12-bit instructions. It is possible to express any function as a sequence of such instructions.

For example, consider the function \( \text{G-form}[x] \) given above. Let \( M_16, M_17, M_18 \) and \( M_19 \) be the machine functions \( \text{equals}, \text{optr}, \text{op} \) and \text{note-error} respectively and let \( M_20 \) be a machine function that we will call \( \text{list} \), that has two arguments and forms them into a list of two items. Also let the instructions that give the atoms \( I, G \) and \( \text{Float} \) be \( C_0, C_1 \) and \( C_2 \).

Then \( \text{G-form}[x] \) can be expressed as the following sequence of instructions.

\[
\begin{align*}
\text{X1 M17 C0 M16 J6 C2 X1 M18 M20 E1} \\
\text{X1 M17 C1 M16 J3 X1 M18 E1 D2 M19 E1}
\end{align*}
\]

Suppose that \( \text{G-form} \) is called in by the instruction \( F_2 \) in the function \( \text{assign} \) and that its argument is the list \( (G, (C, a^*)) \). Then immediately before \( \text{G-form} \) is called in the top item of the Working Store will be a word addressing the list \( (G, (C, a^*)) \) which will have been built up in the Free Store. Let us also assume that the value of the WSP (Working Store Pointer) is \( k \) at this time.

Then when \( \text{G-form} \) is called in a link is placed on the Working Store and the WSP is increased to \( k + 1 \). The significance of such a link will be explained below. Control is then passed to the first instruction of \( \text{G-form} \). The function will be evaluated as follows.

\[
\begin{align*}
\text{X1} & \quad \text{A copy of the argument of the function, i.e.} \\
& \quad \text{the address of the list} (G, (C, a^*)) \text{, is placed on} \\
& \quad \text{the Working Store. The WSP is increased to} \\
& \quad k + 2.
\end{align*}
\]

\[
\begin{align*}
\text{M17} & \quad \text{The Machine function \text{optr} is called in. This} \\
& \quad \text{replaces the list at the top of the Working Store} \\
& \quad \text{by its first item, i.e. the atom} G.
\end{align*}
\]

\[
\begin{align*}
\text{C0} & \quad \text{The atom} I \text{ is placed on the Working Store and the} \\
& \quad \text{WSP is increased to} k + 3.
\end{align*}
\]

\[
\begin{align*}
\text{M16} & \quad \text{The machine function \text{equals} is called in. It} \\
& \quad \text{examines its arguments which are the second} \\
& \quad \text{and first items of the Working Store, and because} \\
& \quad \text{they are different overwrites the first of} \\
& \quad \text{the arguments by the atom \text{False}. The WSP is} \\
& \quad \text{reduced to} k + 2 \text{ so as to point to this atom.}
\end{align*}
\]

\[
\begin{align*}
\text{J6} & \quad \text{The first item of the Working Store is the atom} \\
& \quad \text{False, so a jump of six 6-bit characters is made.} \\
& \quad \text{C2 occupies 12 bits. X1, M18, M20 and E1 each} \\
& \quad \text{occupy 6 bits. So the jump is made to the} \\
& \quad \text{instruction X1 that follows E1. The WSP is} \\
& \quad \text{reduced to} k + 1.
\end{align*}
\]

\[
\begin{align*}
\text{X1} & \quad \text{A copy of the address of the list} (G, (C, a^*)) \text{ is} \\
& \quad \text{placed on the Working Store. The WSP becomes} \\
& \quad k + 2.
\end{align*}
\]

\[
\begin{align*}
\text{M17} & \quad \text{\text{Optr} is again called in, and the word at the} \\
& \quad \text{top of the Working Store is replaced by the} \\
& \quad \text{atom} G.
\end{align*}
\]

\[
\begin{align*}
\text{C1} & \quad \text{Another copy of the atom} G \text{ is placed on} \\
& \quad \text{the Working Store. The WSP is increased to} k + 3.
\end{align*}
\]

\[
\begin{align*}
\text{M16} & \quad \text{The machine function \text{equals} this time finds} \\
& \quad \text{that both its arguments are equal to} G \text{ and so} \\
& \quad \text{replaces the first of these arguments by the atom} \\
& \quad \text{True. The WSP is reduced to} k + 2 \text{ so as to} \\
& \quad \text{point to this atom.}
\end{align*}
\]

\[
\begin{align*}
\text{J3} & \quad \text{The first item of the Working Store is the atom} \\
& \quad \text{True, so no jump is made. The WSP is reduced to} \\
& \quad k + 1.
\end{align*}
\]
Another copy of the address of the list \((G, (C, a^*))\) is placed on the Working Store. The WSP becomes \(k + 2\).

The machine function \(\text{op}\) is called in. This replaces the first item of the Working Store by the most-significant half of the word addressed by the least-significant half of the word addressed by that item. In this case the address pointing to the list \((G, (C, a^*))\) is replaced by an address pointing to the sub-list \((C, a^*)\). The WSP is unchanged.

Terminate the function. The result, which is the address of the list \((C, a^*)\) at the top of the Working Store, overwrites the original argument of \(G\), i.e. the word on the Working Store before the link. The WSP is reduced to \(k\) to point to this word. Control is returned to the instruction in \(\text{assign}\) that comes after the instruction \(F2\).

A more comprehensive list of the instructions that we define to denote various functions, machine functions and constants is now given:

Functions:

\[
\begin{array}{ll}
F0 & \text{assign} \ [x;y] \\
F1 & \text{I-form} \ [x] \\
F2 & \text{G-form} \ [x] \\
F3 & \text{copy} \ [x;y;S] \\
F4 & \text{store} \ [x;S] \\
F5 & \text{instr A} \ [f; x;y;z;S] \\
F6 & \text{instr B} \ [f; x;y;z;S] \\
F7 & \text{opco} \ [x] \\
F8 & \text{store A} \ [S;s;x] \\
\end{array}
\]

Machine Functions:

\[
\begin{array}{ll}
M16 & \text{equals} \ [x;y] \\
M17 & \text{optr} \ [x] \\
M18 & \text{op} \ [x] \\
M19 & \text{note-error} \ [x] \\
M20 & \text{list} \ [2 \ [x;y]] \\
M21 & \text{and} \ [x;y] \\
M22 & \text{atom} \ [x] \\
M23 & S_0 \\
M24 & \text{accept 2} \ [f; x;y;S] \\
M25 & \text{rem} \ [x] \\
M26 & \text{allocate} \ [x] \\
M27 & \text{same} \ [S] \\
M28 & \text{next} \ [S] \\
\end{array}
\]

Constants:

\[
\begin{array}{ll}
C0 & \text{I} \\
C1 & \text{G} \\
C2 & \text{Float} \\
C3 & \text{Int} \\
C4 & \text{C} \\
C5 & +_G \\
C6 & -_G \\
C7 & \times_G \\
C8 & /_G \\
C9 & +_I \\
C10 & -_I \\
C11 & \times_I \\
C12 & -47 \\
\end{array}
\]

A machine function \(S_0\) is listed instead of a constant \(S_0\) since the function \(S_0\) has to construct a word on the Free Store indicating the first accumulator available for intermediate results. The result of the function is the atom \(S_0\) which points to this word. When a copy is made of this atom, the copy will point to the same word on the Free Store. Then if the machine function \(\text{next}\) is applied to any copy of the original word \(S_0\), the word on the Free Store will be altered, and all the copies of \(S_0\) will be affected and become \(S_1\). The effect of the machine function \(\text{same} \ [S]\) is to construct another copy of the word on the Free Store, and so create a copy of \(S\) which will not be affected by applications of the function \(\text{next}\) to a copy of the original \(S\).

Some further examples of functions expressed as sequences of instructions are now given:

Assign \([x;y]\) becomes

\[
\begin{array}{ll}
X1 & M17 C0 M16 X1 M18 M17 C4 M16 M21 J0 \\
X2 & F1 X1 M18 M18 M23 F3 E2 \\
X1 & M17 C1 M16 X1 M18 M17 C4 M16 M21 J0 \\
X2 & F2 X1 M18 M18 M23 F3 E2 D1 M19 E2 \\
\end{array}
\]

Copy \([x;y;S]\) becomes

\[
\begin{array}{ll}
X3 & M22 J7 D12 X2 X3 X1 M24 E3 \\
X3 & M17 C4 M16 J8 D4 X2 X3 M18 X1 M24 E3 \\
X3 & M17 C3 M16 J4 D8 X3 M18 X1 F4 C12 \\
X2 & X1 F6 E3 \\
X3 & M17 C2 M16 J4 D8 X2 M18 X1 F4 D47 \\
X2 & X1 F6 E3 \\
X3 & M17 C5 C6 C7 C8 A4 J20 \\
X3 & M17 F7 X3 M18 X1 F4 X3 M25 M18 X1 \\
F4 & X2 X1 F6 E3 \\
X3 & M17 C9 C10 C11 A3 J14 \\
X3 & M17 F7 X3 M18 X3 M25 M18 X2 X1 F5 \\
E3 \\
D4 & M19 E3 \\
\end{array}
\]

In the above function \(2nd \ \text{op} \ [x]\) is expressed as \(\text{op} \ [\text{rem} \ [x]]\). Also the instruction code have been expressed as the decimal values they have in Orion machine instructions. An instruction code written as \(mn\), where \(n\) is the last digit and \(m\) represents the others is stored in Orion as \(8m + n\).

Store \([x;S]\) becomes

\[
\begin{array}{ll}
X2 & M22 J3 X2 M26 E2 \\
X2 & M17 C4 M16 J3 X2 M18 E2 \\
X1 & M27 X1 M28 X2 F8 E2 \\
\end{array}
\]

Entry to and exit from a function

There are two registers known as CONTROL and INTLINK. At any time CONTROL contains the address of the next 6-bit or 12-bit instruction that the Interpreter is going to interpret. INTLINK contains the address of the last link that was placed on the Working Store.

The contents of CONTROL and INTLINK are both 24-bit quantities and so they can both be stored in one Orion word. When the Interpreter obeys an
instruction of the type \( F_n \), the contents of these two registers are put together and placed as a new word on the Working Store. This word is the link for the new function. The address of this word is then placed in INTLINK and the address of the first instruction of the new function is placed in CONTROL. The Interpreter can then proceed to obey the function.

Conversely, when the Interpreter obeys an instruction of the type \( E_n \), the link whose address is in INTLINK is recovered. From this word are obtained the previous values of the contents of CONTROL and INTLINK and these values are restored. The result of the function is placed on the Working Store \( n \) words below the link, so that the first argument of the function is replaced. In the case of \( E_0 \) the function has no arguments and the link itself is replaced by the result. The interpreter then continues to interpret the original function.

Suppose the Interpreter is obeying a function \( F_1 \) which was called in by \( F_2 \) which itself was called in by \( F_3 \). Then the system of links is as shown in Fig. 5.

The function input routine

It would have been possible to write a routine to read in functions described in the original functional notation. Since, however, it was not difficult to write the functions as a sequence of instructions as above, the input routine was written to accept functions in this form.

One refinement in this input routine allows the instruction \( N_n \) to be written to represent the Orion instruction code \( n \), and so save the trouble of converting the codes to their decimal equivalents.

The greatest deficiency in the input routine is that it requires the conditional jump instructions to be written specifying the number of characters to be jumped. This has proved to be a source of errors, and it would clearly have been worth while to have written the input routine to allow jumps to labelled instructions. Apart from this, however, it has proved to be quite convenient to input functions as sequences of instructions.

The input routine packs the functions one after another in the store, and creates a table of the addresses of the first instruction of each function. The first instruction of a function does not have to occupy the initial position of a 48-bit Orion word, but follows immediately after the last instruction of the previous function. Also, a 12-bit instruction can be split between two words occupying the least-significant 6 bits of one word and the most-significant 6 bits of the next word. In this way the maximum packing density is achieved.

The working of the interpreter

When the Interpreter obeys an instruction it first switches on the first of the 6-bit characters by jumping to the address that is stored in the appropriate word of a 63-word Switch Table. If the value of the character is \( n \), a jump will be made to the address in the \((n + 1)\)th word of the table. At this address is a routine that will deal with the instruction, if it is a single-character instruction. If the instruction consists of two 6-bit characters, the routine at this address will read the second character and will act according to the specification of the complete 12 bits.

For example, suppose that the value of the first character is in the range 0 to 4. This signifies an instruction of the type \( F_n \), and the first 5 words of the Switch Table each contain the address of the routine that deals
with such instructions. This routine extracts the next 6-bit character and this, together with the first character will give an integer in the range 0 to 319. According to the value of this integer the routine will call in one of up to 320 possible functions. It will obtain from the table of addresses of the functions the address of the first instruction of the required function. The contents of CONTROL is set equal to this address and the function entered.

The significance of other values of the first character of an instruction are as follows:

5   This signifies a machine function Mn where 64 < n < 127. The particular machine function is determined by the next 6-bit character.

6   An instruction of the form Xn where n ≥ 6. The next character gives the value of n.

7   An instruction of the form En where n = 0 or n > 6. The next character gives the value of n.

8–11 An instruction of the form Cn. With the next character an integer in the range 0 to 255 is determined which specifies the required constant.

12–15 An instruction of the form Jn. With the next character an integer in the range 0 to 255 is determined which specifies the number of characters to be jumped if the condition for jumping is satisfied.

16–27, 30–49, 52 and 53 Each signifies a single-character machine function of the form Mn. The value of the character gives the value of n.

28   An instruction of the form An where n > 3. The next character gives the value of n.

29   This signifies the single-character instruction A2.

50 and 51 An instruction of the form Dn. With the next character an integer in the range 0 to 127 is determined.

54–58 Single-character functions signifying XI to X5, respectively.

59–63 Single-character functions signifying E1 to E5, respectively.

It can be seen from the above arrangement that the number of functions, machine functions and constants allowed is limited. However, this arrangement gave us all that we needed for the EMA Compiler. If necessary the ranges could have been increased by reducing the number of single-character instructions.

The function assign would be stored in the computer as the following sequence of 6-bit characters packed eight to a word.

\[
\begin{align*}
54, & 17, & 8, & 0, & 16, & 54, & 18, & 17 \\
8, & 4, & 16, & 21, & 12, & 10, & 55, & 0 \\
1, & 54, & 18, & 18, & 23, & 0, & 3, & 60 \\
54, & 17, & 8, & 1, & 16, & 54, & 18, & 17 \\
8, & 4, & 16, & 21, & 12, & 10, & 55, & 0 \\
2, & 54, & 18, & 18, & 23, & 0, & 3, & 60 \\
50, & 1, & 19, & 60
\end{align*}
\]

This uses up \(6\frac{1}{2}\) words, and if we also take I-form, G-form, copy and store, the five functions together occupy only 33 words. This illustrates the compactness of the language.

Conclusion

Looking back I feel sure that the use of this list-processing language benefitted the project as a whole. The instruction code of Orion was very suitable for implementing the Interpreter, particularly as regards the instructions for handling characters which provided an easy way of obtaining access to the packed 6-bit and 12-bit instructions. The language was easy to use, it proved both powerful and compact, and development proceeded quickly.

The main disadvantage was the effect on the speed of the compiler. The compiler could doubtless have been written completely in machine orders, and this would have resulted in a faster compiler. Such a compiler would, however, have occupied a lot more space and would probably also have taken considerably longer to write and develop.

Acknowledgements

I wish to acknowledge the part played in this project by Mr. P. J. Landin who did much of the initial work, and also the assistance of my colleague Mr. D. J. M. Martin.

This paper is published by permission of International Computers and Tabulators Limited.

References
