MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 179

August 1969

The arithmetic-statement pseudo-ops: .I & .F

B.K.P. Horn

This is a feature of MIDAS which facilitates the rapid writing and debugging of programs involving much numerical calculation. The statements used are ALGOL-like and easy to interpret.

An arithmetic-statement expander:

Since the Incompatable Timesharing System (ITS) does not support an ALGOL style compiler and LISP is so cumbersome in dealing with arithmetic statements, it is very tedious to perform even the simplest algorithms of numerical analysis. To alleviate this problem without an inordinate amount of effort, two pseudo-ops where added to MIDAS (The macro-assembly language).

The pseudo-ops are .F and .I. The first of these will have the arithmetic in the arithmetic statement following it performed in floating point, the latter in fixed point.

Each statement is treated without reference to any of the others. Spaces may appear in a statement almost everywhere and are ignored. Exceptions are in the continue part of a continuation statement and in a subscript.(See later on)

Arithmetic statements are combinations of variable names, numbers, function names and operators. Normally each statement specifies the calculation of one or more values and where they are to be stored.

The operators are:

= ( ) < > ↑ / * + - ## $,

A number is a character-string starting with a numeric character (0,1 ... 9) followed by non-operators. This number should make sense to MIDAS. The operator ↑ is permitted to appear in the number, being the seperator used in MIDAS for the exponent of a number.

A variable (or function) name is a character-string starting with a character which is neither numeric nor an operator and consists of up to six non-operators.

/ * + - have the usual meaning of divide, multiply, add and subtract. ↑ is used for exponentiation. (and) are used to force the precedence as usual, ie. normal evaluation proceeds from left to right, with exponentiations being performed first, then multiplications and divisions, and additions and subtractions last, except that expressions in parentheses are evaluated first. This is strictly adhered to and thus A↑B↑C≡(A↑B)↑C unlike the FORTRAN convention A**B**C≡A**(B**C). Nested pairs of parentheses are evaluated from the inside out.

Intermediate results are kept in a stack which has to be in the accumulators and is defined by the user. These accumulators are called AØ,A1 ... A9. If fixed point arithmetic is used Ai ≠ Aj + 1 if i < j . Most commonly AØ == 1, A1 == 2 etc. Usually only the first few accumulators in the stack are used.

< and > surround the arguments of a function. The arguments are seperated by ,'s. Thus a name as defined above is a function name if it is followed by a <. For example:

MAX< A-B*CD,23.4> and RANDOM<>

If used not directly following a name, <&> act exactly like (and).

Functions return a single value in AØ. The assembled code includes a PUSHJ P, to the function, the user being responsible for providing a subroutine which

accepts the arguments as presented in A0, A1 etc., does not disturb any accumulators other than those in which the arguments where passed and returns the result in AØ before executing a POPJ P, .

A variable name followed directly by a (is considered to be a vector. The subscript between the (and the matching) can of the following form:

AC ± NUM
± NUM + AC

Where AC is the variable name of an accumulator in which the subscript is assumed to have been loaded. NUM is a number, acting as a displacement.

= indicates that the value available at this point (as calculated by the portion of the arithmetic statement to the right) is to be stored as the value of the variable name to its left. More than one = may thus appear in one arithmetic statement. For example:

.F A=B=ARM-LOSS=FOO*BARF

This invokes the multiplication of FOO by BARF, storage of the result in LOSS. Next LOSS is subtracted from ARM and the result stored in both A and B. More complicated constructs are possible by making use of parentheses. Some care is required in arranging the right sequence of storage operations so as not to overwrite values needed further on. (Perhaps a more intuitive structure could be given to multiple equals if one did not adopt the FORTRAN like convention of having the statement <u>follow</u> the equals)

permits the passing of arguments by name rather than by value, ie. is it performs a quoting action. This is particularly useful for subroutines operating on vectors (Dotproduct for example), or subroutines executed for their effect rather than their value. It also permits the passing of a function address as an argument. This is achieved by surrounding the variable name with [and] .

$ indicates a continuation and must be directly followed by a 'return carriage', 'line feed'(usually supplied by TECO anyway) and either .I or .F (which is ignored) a space or tab and the continuation of the statement. For example:

```
.F ANSWER=273.0/T $
.F -IN*VEST*MENT
```

Unitary + and * are ignored. Unitary / and - are interpreted as 1.0/ and 0.0- respectivly. = and ↑ may not appear unitarily.

Since @ and ' may be part of a variable name, one can make full use of MIDAS's indirect addressing and automatic variable storage assignment conventions. The use of @ comes in very handy when working with multi-dimensional arrays addressed through margin-arrays.

↑ normally generates a call to a function called EXPLOG, which gets two arguments. To facilitate generation of fast inline exponentiation one may follow the ↑ directly by the single digits 1,2,3 or 4. For example:

```
.F R=SQRT<X↑2 + Y↑2>
```

Some unusual constructs are possible which are usually interpreted as one would intuitivly expect. For example no equals sign need to appear in an expression, the result is then merely left in AO. Next the comma may be used to force evaluation to take place in higher slots in the stack AØ,A1 ... A9 . So for example:

```
.F A*Z,LENGTH,SQRT < 365.4/XSS > ,ABEL/CAIN
```

is interpreted as meaning the calculation of the four specified expressions and leaving the results in AØ, A1, A2 and A3.

```
.F ,SAVA'=,SAVB'=,SAVC'=
```

Here AØ will be stored in SAVA, A1 in SAVB etc.

```
.I DISLIS=#HCKLST,A=B=C=O,QUARK=KAESE
```

is just a convenient way of combining four operations in one statement.(At this stage)= has a higher weight than , and so the first expression will be carried out in AØ, the second in A1 etc. with storage of results starting at QUARK and ending at DISLIS.

The statement expander is not optimal in its use of accumulators, thus small savings in program length and execution speed can be had by writing the precedence-forcing parentheses in the right order. For example:

```
.F Y=(((A*X+B)*X+C)*X+D)*X+E
```

uses one accumulator in the stack and the minimal number of instructions, while:

```
.F Y=E+X*(D+X*(C+X*(B+A*X)))
```

uses eight accumulators in the stack and as many extra MOVE instructions - although the accumulator to accumulator instructions used are slightly faster than the storage to accumulator instructions used in the first example. These examples are rather extreme however and it hardly ever pays to worry about such minor questions of efficiency.

To use the statement expander on merely uses TECO as usual to create a MIDAS program containing the required pseudo-ops - remembering to define AØ,A1 etc. and the push-down accumulator P. During the subsequent assembly with MIDAS, error messages concerning the arithmetic-statement pseudo-ops are presented in an unusual manner. The offending statement is typed, with a questionmark where the error was first noticed and an explanation of the error on the next line. No assembled output is generated for erroneous statements.

The standard functions SIN, COS, ATAN, LOG, EXP, SQRT, ABS, EXPLOG, FRACT, INTEG, MOD, RANDOM can be found in the file SUPPRT ROUT (on DSK: BKPH;). This file also contains floating point input and output routines to teletype and printer. Other features are interupt-handling, control character features and a simple command language. Most of these features can be exised selectivly, using switches on the first page of the program. This program is intended to be inserted by means of the .INSRT pseudo-op and consists of bits and pieces high-jacked from L.J.Krakauer, T.Binford, C.Nelson, R.Greenblat and others.