66-6 Cop. 2

X

PRELIMINARY DESIGN SPECIFICATIONS OF XX, A VARIANT OF L^6.

J. Earley and A. Newell

July 13, 1966

This is a working draft and may not be cited or reproduced without the permission of the authors. The research was supported by Contract SD-146 from the Advanced Research Projects Agency of the Department of Defense.

Carnegie Institute of Technology

7/13/66

PRELIMINARY DESIGN SPECIFICATIONS OF XX, A VARIANT OF L^6.

J. Earley and A. Newell

This paper describes a variant of L^6* which is being implemented for the IBM 360 computer at Carnegie Tech. The language (as yet unnamed and simply called XX here) is embedded in the OS 360 Assembler Language and consists of a series of macros which may be called by the source language programmers. This accounts for some of the unusual and limited syntax.

XX is intended to be a system in which list processing systems and languages may be written making it unnecessary for the systems programmer to go to the machine level. However, it may also be used effectively as a programming language itself. The aim is to give the programmer as much control as possible over all the processes he may use -- available space, list linking, block storage allocation -- while maintaining reasonable programming ease.

Sections I and II contain the definitions and basic operations which make up the heart of XX. Section III contains additional operations necessary to make it complete as a programming language. This much represents Basic XX. With this language one should be able to accomplish any of the operations intended for the language, though maybe not easily or efficiently. Section IV contains certain operations which are merely abbreviations for sequences of Basic XX commands, but which make some very

---

* Knowlton, Kenneth C., A Programmer's Description of L^6, Bell Telephone Laboratories' Low-Level Linked List Language. Bell Telephone Laboratories [illegible] 1966.

common operations easy to write. Section V defines a language in which a programmer may communicate to the XX compiler how he would like certain operations to be implemented, and Section VI defines a similar language in which certain monitoring checks may be turned on.

A general familiarity with Knowlton's L^6 system is assumed in this paper.

I. Definitions

A. Blocks

<Block Definition> ::= BLOCK <Identifier>, (<Number>, <Number>)

<Identifier> ::= <Letter> | <Identifier> <Letter> | <Identifier> <Digit>

<Number> ::= <Digit Sequence> | <SETA Symbol>

A block is a number of contiguous words in memory which are to be treated as a unit. In a block definition, a name is defined to stand for a specific type of block. This is designated by the numbers of the beginning and end words of the block. These may be used to select a sub-block of a containing block or to define the extent of a block.

Thus after executing

```
BLOCK B, (4, 10)
```

we may use B to refer to words 4 through 10 inclusive of a larger block. After executing

```
BLOCK SPACE, (0, 1023)
```

we may use SPACE to set up a block of 1024 words at some location. The initial word of a block is called the 0th one for numbering purposes.

The extent of a block may be defined by using decimal numbers or SETA symbols (see Assembler manual) which hold compile-time arithmetic quantities. Thus, although blocks may be redefined, they are static at run-time in the sense that the programmer always knows for any particular use of a block name the size and shape of the block to which he is referring. This philosophy has been used in the design of the language in that no tests are included to determine the shape of a block or its location in a larger block

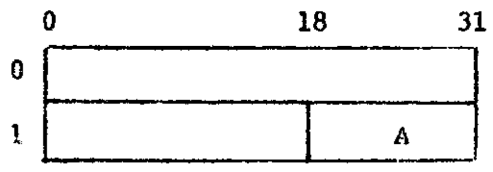
B. Fields

<Field Definition> ::= FIELD <Identifier>, <Number>, (<Number>, <Number>)

A field is a number of contiguous bits within a word of a block which are to be treated as a unit. A name is defined to stand for a specific type of field in a field definition. The first number gives the word of the block in which the field resides. The next two define the boundaries of the field just as in a block definition. Thus,

FIELD A, 1, (18, 31)

sets up field A to be this part of a two word block:



Field definitions may be used with any block. Like blocks, they may overlap or coincide in any way, and one may be accessed from a block of any size, including an implied block (see IIA). Like blocks, field shapes are static at run-time and so are known to the programmer.

C. Bugs

<Bug Definition> ::= BUG <Identifier List>

<Identifier List> ::= <Identifier> | <Identifier List> <Identifier>

A bug refers to a specific location in memory, unlike blocks and fields, which are only types, to be referenced relative to the beginning of a block. A bug is a full word in length and like fields and blocks it may contain any information the programmer chooses to store in it. Bugs participate in all the same operation as do full word fields.

In a bug definition, each name is declared to stand for a specific bug. Its location is assigned by the compiler.

II. Basic Operations on Fields and Blocks

A. Designation

<Elementary Identifier> ::= <Letter> | <Elementary Identifier> <Digit>

<Name> ::= <Elementary Identifier> | [<Identifier>]

<Sequence> ::= <Name> | <Sequence> <Name>

<Field Name> ::= <Name>

<Field Sequence> ::= <Sequence> <Field Name>

<Block Name> ::= <Name>

<Block Sequence> ::= <Sequence> <Block Name>

The first name in the sequence and only the first must be a bug name. Each field is assumed to contain a pointer to a location in memory. An implied block is assumed to exist there with that location as its 0th word. If a field is too small to contain an address, 0's are added on its left to make it the right size.

The value of the sequence is calculated in the following way: We start by accessing the implied block pointed to by the initial bug. If the next name is a block name B, we then access the B sub-block of that block. If the next name is a field name F, we access the F field of the block and then access the implied block pointed to by that field. In either case we now have a block again, so the process is repeated. This continues through the sequence until the last name gives us either a block or field as a value. In the case that a field is last, we do not access its implied block.

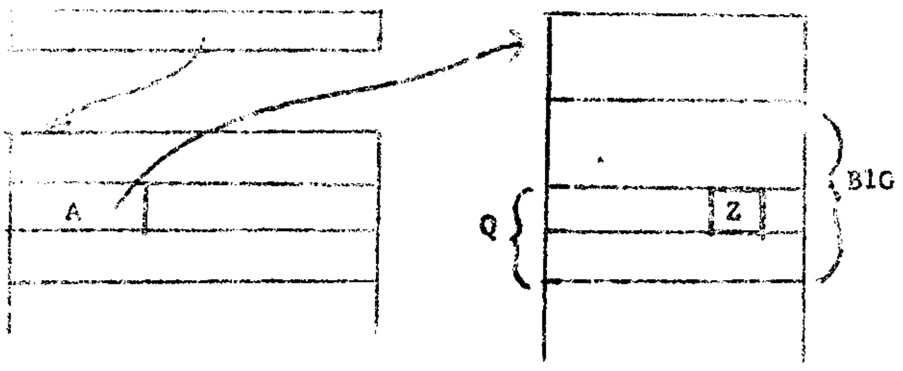
Thus to calculate

ZA[BIG,Q]

where BIG W, [illegible]

```
FIELD A, 1, (0, 15)
FIELD Z, 0, (21, 26)
BLOCK BIG, (1, 5)
BLOCK Q, (2,3)
```

we go through the following chain:



B. Basic Field Operations

```
<Field Operation> ::= (<Field Sequence>, ←, <Quantity>)
                      (<Field Sequence>, ←0, <Sequence>)
<Quantity> ::= <Field Sequence> | <Number> | $ <Expression>
```

A quantity may be the contents of a field, a decimal number, or any legal assembler expression preceded by a $.

"←" stores the right quantity into the field on the left right-justified. If the quantity is too small, the rest of the field is cleaned to 0. However, if the quantity is too large it may clobber any information stored to the left of the field.

"←o" makes the field on the left point to the block indicated by the sequence on the right. If it is a block sequence, the left field points to the block which is its value. If it is a field sequence, the left field points to the implied block of the field which is the value of the sequence.

C. Basic Block Operations

```
<Block Operation> ::= (<Block Sequence>, ←, <Block or Quantity>)

<Block or Quantity> ::= <Block Sequence> | <Quantity>
```

When a quantity is stored into a block it is placed right-justified in the last word of the block and the rest of the block is cleared to 0. If a smaller block is stored into a block it is placed in the last words and the rest is cleared to 0. If a larger block is stored it may clobber contiguous information.

D. Basic Tests

```
<Basic Test> ::= (<Sequence>, <Equals>, <Block or Quantity>) |
                 (<Field Sequence>, <Points>, <Sequence>) | <Sequence> |
                 ¬<Sequence>

<Equals> ::=  = | ¬=

<Points> ::=  =o | ¬=o
```

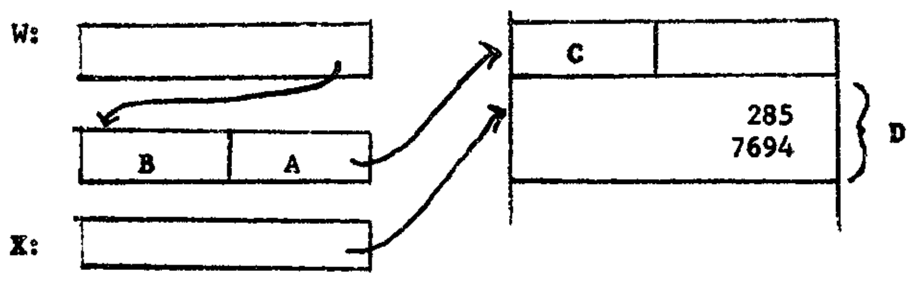
One may test for equality or inequality between the contents of a field or block and a quantity or block or field contents. If the two operands do not match in size, leading 0's or leading blank words are added to the smaller one to make it conform before the test is made.

A field may be tested to see whether or not it points to the implied block of some field sequence or the value of a block sequence.

If just a field or block sequence appears, the test is true if it is non-zero and false otherwise. "¬" reverses these.

Examples:

Suppose we have



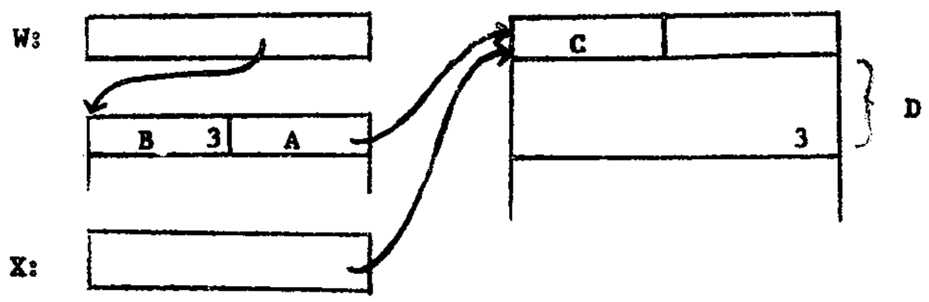
(X, =o, WAD) is true

(X, ←o, WA) makes X point where A points

(WB, ←, 3) puts a 3 in B

(WAD, ←, 3) puts a 3 in D and clears the rest of it.

Now we have



Now (WB, =, XD) is true

III. Additional Programming Features

Through the use of defined blocks and fields, one may access sub-blocks and fields which are fixed in their location add size, but which may lie in varying blocks. If one needs to access varying sub-blocks or words of a block, he may perform arithmetic on the field pointing to the block to obtain the needed result. If one needs to access varying fields within a word he may perform logical operations on the word to obtain the result.

A. Arithmetic

```
<Arithmetic Operation> ::= (<Field Sequence>,
                <Arithmetic Operator>, <Quantity>)

<Arithmetic Operator> ::= + | - | * | /
```

The contents of the field which is the value of the field sequence is modified by executing the indicated operation between it and the quantity, and storing the result back into the field as in an ordinary store command.

Thus (XY, +, 3) increments the contents of the Y field of X by 3. In all fields except those which are full words, numbers should be kept to non-negative integers. If any operation produces a negative number, garbage may result. In full word fields, signed arithmetic will work correctly.

```
<Arithmetic Test> ::= (<Quantity>, <Relation>, <Quantity>)

<Relation> ::= > | ≥ | < | ≤
```

These tests treat their operands in the same form as the basic tests.

B. Logical Operations

```
<Logical Operation> ::= (<Field Sequence>, <Logical Operation>,
                        <Quantity>) | (¬, <Field Sequence>) |
                        (<Field Sequence>, <Shift Operator>, <Quantity>)

<Logical Operator> ::= ∧ | ∨ | x ∨

<Shift Operator> ::= ↞ | ↠
```

The logical operators modify the field on the left in a way analogous to arithmetic operators. Thus (X, ∨, Z) stores into X the logical union or X and Z. "X∨" executes the "exclusive or" operation. "¬" complements the contents of the field which is its argument.

The shift operators shift the contents of the left field left or right the number of bits specified by the quantity on the right. This shift is done within the field itself. 0's are supplied where bits are vacated by the shift, but any bits which overlap after the shift may cause trouble.

C. Input/Output and Conversion

<I/O Operation> ::= (<Sequence>, <Device>, <Quantity>) | (<Device>, NEXT)

<Device> ::= PRINT | PUNCH | READ

These operations designate that the number of characters specified by the number quantity on the right are to be read into or taken out of the field or block specified on the left. As with other operations, this is done right-justified and no check is made for overlap on the left. (Some operations for conversion of digits from card code to binary are also needed.)

D. Control Structure

```
<Control Operation> ::= (GOTO, <Identifier>)
                        (CALL, <Identifier>)
                        (GOTO, RETURN)
```

GOTO specifies a simple change of control.

CALL produces a call on a closed subroutine starting at the specified label.

RETURN specifies that control is to return to the next statement after the most recent call on a subroutine.

```
<Operation> ::= <Field Operation> | <Block Operation> |
<Arithmetic Operation> | <Logical Operation> |
<Control Operation> | <I/0 Operation>

<Test> ::= <Basic Test> | <Arithmetic Test>

<Operation Sequence> ::= <Operation> | <Operation Sequence> <Operation>

<Test Sequence> ::= <Test> | <Test Sequence> <Test>

<Definition> ::= <Block Definition> | <Field Definition>|<Bug Definition>

<IF Word> ::=  IF | IFANY

<Statement> ::= <Definition> | <IF Word> <Test Sequence>, THEN,
                <Operation Sequence> |
                DO <Operation Sequence>
```

DO specifies that each if a sequence of operations are to be executed in order. IF specifies that if all of the tests in the sequence are true then all the operations following the THEN are to be executed. IFANY specifies that if one or more of the tests are true, then all the operations are to be executed.

```
<Labeled Statement> ::= <Identifier> <Statement> | <Statement>
```

A program is a series of labeled statements embedded in an appropriate assembler language program.

IV. Abbreviations

The following operations are in the spirit of XX, but not required as primitive definitions. Therefore they are included here along with the Basic XX code which defines them.

A. Push and Pop

<Push> ::= ↓<Field Sequence> <Field Name>

<Pop> ::= ↑<Field Sequence> <Field Name>

<Left Side> ::= <Push> | <Field Sequence>

<Right Side> ::= <Pop> | <Quantity>

<Push-Pop Operation> ::= (<Left Side>, ←o, <Right Side>)

Push and pop work on a list of blocks. The field sequence following the "↓" or "↑" points to a certain block in the list. The field name specifies the field by which the blocks of the list are linked.

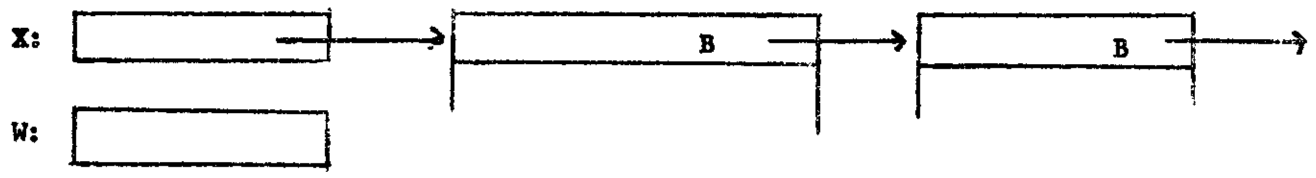
The pop operation deletes the block from its list and points to it as indicated on the left. Thus,

(W, ←o, ↑XB)

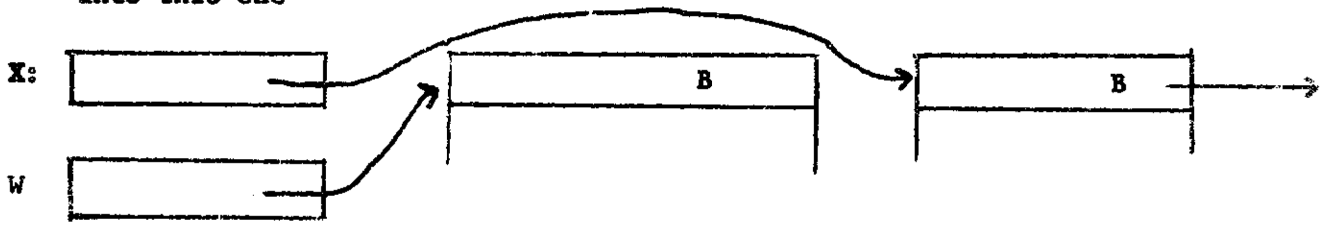
is equivalent to

(W, ←o, X), (X, ←o, XB)

It changes this configuration



into this one



The pop operation may be used as in an ordinary stack when its field sequence points to the top of the stack. It may be used to delete an element from the middle of a list if the field sequence links through the list to the block to be deleted.

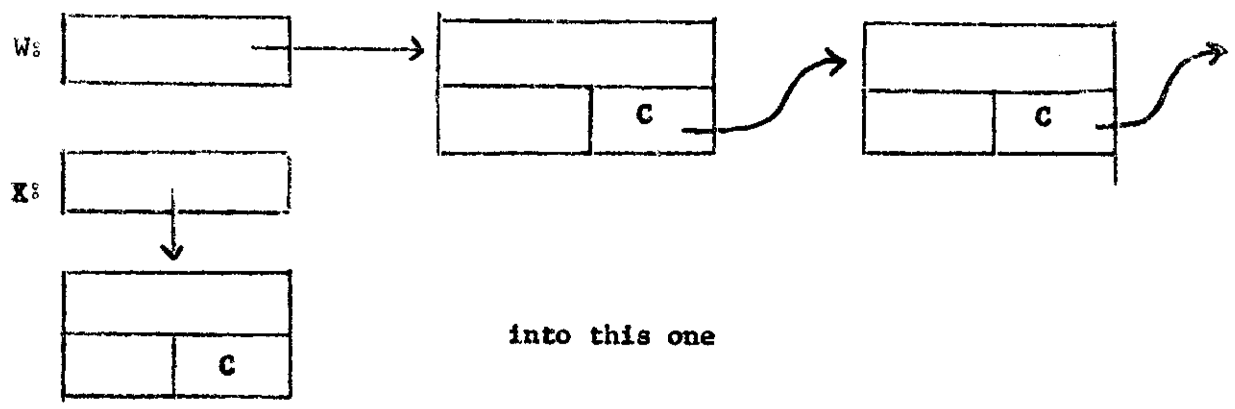
The push operation inserts the block pointed to on the right side into the list on the left so that it is pointed to by the field indicated. Thus,

(WCC, ←o, X)

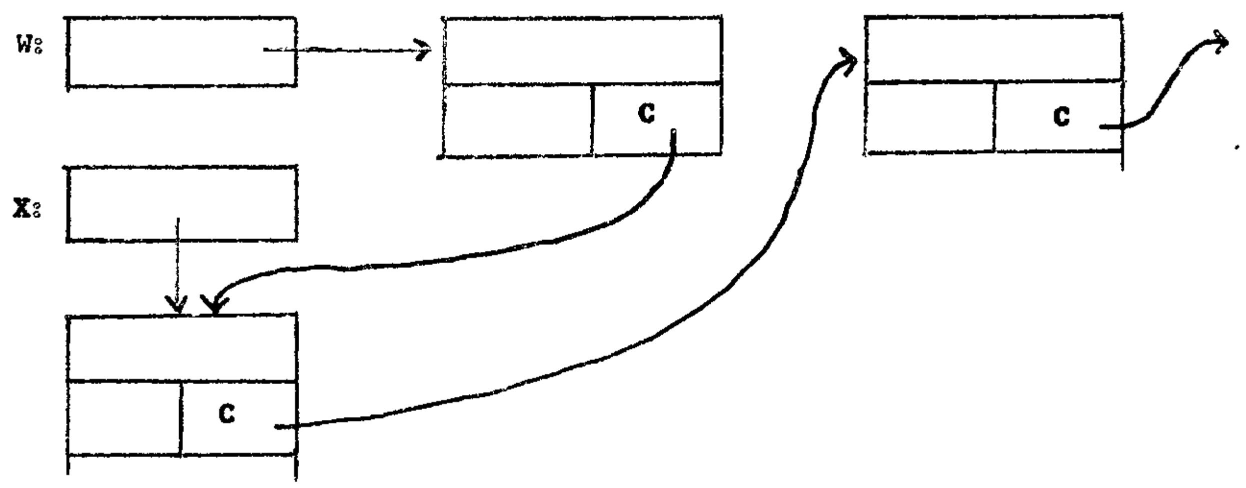
is equivalent to

(XC, ←o, WC), (WC, ←o, X)

It changes this configuration



into this one



Notice that X still points to the inserted block.
These operations may also be combined:

(↓XA, ←o, ↑YB) ≡ (YA, ←o, X), (X, ←o, Y), (Y, ←o, YB)

B. Move Pointer

<Move Pointer> ::= (Field Sequence>, →, <Field Sequence>)

This is an abbreviation for a pointer set operation which moves the pointer down a list. (FA, →, BC) is equivalent to (FA, ←o, FABC)

C. Block Splitting

<Split Operation> ::= (SPLIT, <Block Sequence>, <Quantity>, <Field Name>)

This operation splits the block into sub-blocks of the size indicated by the quantity and links them together using the field indicated. A zero is placed in the last field.

Given the definition

```
        BLOCK    B,(I, J)
```

the equivalent of

```
        (SPLIT, AB, K, F)
```

is the following (where A, R, L, P are bugs, F is a field, and I, J, K represent integers):

```
        DO (R,←o,A), (R, +, I)
        DO (L, ←o, A), (L, +, J)
LOOP    DO (P, ←o, R), (R, +, K)
        IF (R, >, L), THEN, (GOTO, OUT)
        DO (PF,←o, R), (GOTO, LOOP)
OUT     DO (PF, ←, 0)
```

D. Available Space Operations

The block splitting operation gives the programmer a method for setting up lists that could be used for available space lists. He may then use push and pop operations for getting new cells from available space and returning them. In order to make these processes even easier, the following abbreviations are available.

```
    <Available Space> ::= (SPACE, <Field Sequence>, <Number>,
                                  <Field Name>, <Number>)
```

This sets up an available space list in blocks of the size given by the first number linked together by the field name. It is pointed to by the field sequence and it extends the number of words in memory past this point given by the second number.

```
    (SPACE, S, 4, F, SIZE) ≡
     BLOCK B, (0, SIZE)
      DO (SPLIT, SB, 4, F)
```

Since the programmer has supplied the name of the space list and its link field, he can access it using push and pop or perform any other L^6 operations on it. If he wants to move all this behind the scenes, however, he may execute

```
        (SPACE, <Number>, <Number>)
```

This causes an available space list to be built up in blocks of the size of the first number. The second number gives the total size of the list in words. The location and link field of the list are unknown to the programmer. He may now use just "↑" and "↓" with no parameters to work with his available space list. Thus, (W, ←o, ↑) points W to a new cell. And (↓, ←o, ↑WA) pops the top element off WA and returns it to available space.

Even though an available space list S has been defined in this way, another list may have been defined more recently and would therefore be the one evoked by "↑" and "↓". To return this function to S, we execute

(SPACE, S).

V. The following declarations allow the programmer to specify how certain operations are to be implemented.

```
<Implementation Declaration> ::= <Implementation Feature> <Scope>

                                   PRIORITY MASK
<Implementation Feature> ::= PRIORITY BUG
                                   MASK EQUALS <Field Name>
                                   MASK COMPLEMENT <Field Name>

<Scope> ::= DEF| ON | OFF | <empty>
```

The priorty features must have a scope of DEF. They indicate that all bugs or field masks defined in the next definition are to reside in general registers. The two mask features must also have a scope of DEF. They indicate that field masks defined in the next definition are to be the same as the masks of another field or the complement of those of another field with respect to a 32 bit word. For the complement feature to be used, the original field must reside at one end of a word.

VI. The following declarations allow the programmer to declare the certain monitoring and error checking features are to be operating in this program.

```
<Monitoring Declaration> ::= <Monitoring Feature> <Action> <Scope>

<Monitoring Feature> ::= FIELD OVERLAP
                         SHIFT OVERLAP
                         NEGATIVE INTEGER
                         ZERO LINK <Parameter>
                         ACCESS <Parameter>

<Parameter> ::= <Field Name> | <Empty>

<Action> ::= <Control Operation>
```

Each of these declarations causes the compiler to check for the occurrence of certain conditions. When it finds one of these it executes the action which has been specified with it. This must be either a goto or a call on a subroutine. In the case of a goto, control just continues from there. In case of the call of a subroutine, control may return either to the next operation after the one one which it trapped, or it may return and attempt to execute the instruction again.

"Field overlap" checks to see if one is storing a quantity into a field which is too small for the quantity. This check also operates on arithmetic and logical operations which leave their results in fields. "Shift overlap" checks to see if the quantity which was shifted now overlaps its field within its word. "Negative integer" tests to see if an arithmetic operation puts a negative integer into a field which is less than a full word.

"Zero link" checks to see when the program tries to link through a field which contains 0. This may apply only to a specific field if that field is mentioned as the parameter of the feature. "Access" checks every time any field is accessed or every time a specific field is accessed according to its parameter.

The scope of these declarations may be DEF, specifying that they apply to the object defined in the next definition throughout the program. A feature may be turned ON and OFF, or if no scope parameter is given, then it applies to the next statement only.