WIS-CS-74-218

COMPUTER SCIENCES DEPARTMENT
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Received August 7, 1974

AMINOL REFERENCE MANUAL

by

D. R. Fitzwater

and

Donn R. Milton

Technical Report # 218

October 1974

ERRATA

p. 17 line 2 should be
<expression state>::= GP((<generator stack>) (<macro stack>))

line 3 should be
<generator stack>::= (<program level>)|<generator stack>

p. 22 line 5 should be (0) = <object>

p. 30 next to last line:
((X) (((ϕ..61ϕ..1,N%,1%%)) (ϕ..61,(..22,5%,N)),N))))

p. 31 on the right-hand side of line 5
replace 11 with 41

on the left-hand side of line 3,
delete comma after ..61

next to last line, replace DECVAR with DECSYM

p. 35 line 8, left hand side should be ((↑) ( ) ((Y)))
X
fifth line from bottom, left-hand side should be
Y
((↑) ( ) (Y((W))%))

p. 36 fourth line from bottom, replace the last left parenthesis with ϕ

p. 48 line 3, change projected to protected
line 6, add list> to end of line

p. 59 line 13, left-hand column should be (..1,X%,Y%)

in NOTE3, ((W) ((A) (B)))

p. 64 in 2nd and 4th line from bottom change < to ϕ

p. 82 line 18, change 15 to 3

p. 84 replace fifth line from bottom by: sion state is attached -

p. 98 line 9, change in to is
6th from bottom, change butter to buffer

p. 99 line 8, change fied to fies and object to objects

line 9, change specified to specifies

p. 103 6th line from bottom, change is to i.e.

5th line from bottom, change opa to ops

ABSTRACT

AMINOL is a high-level machine language for creating and managing virtual processes in a network of multi-processing systems. It is a language with which logical operating system functions may be factored and safely delegated among a hierarchy of processes. Facilities are provided to enable processes to create resources, define resource access mechanisms, and allocate these down the process tree with protection implicitly guaranteed.

AMINOL possesses the necessary quality for any machine language in that every possible program stream is syntactically well-formed and semantically well-defined. In addition there are capabilities for operations on structured objects, for the introduction of typed values, and for definitional extension.

AMINOL REFERENCE MANUAL

Table of Contents

1. INTRODUCTION

AMINOL is A Machine Independent Network Operating Language. It is a high level language for creating and managing virtual operating environments for multiple processes in a network of virtual multi-processor systems. In effect, it is the machine language that is common to all virtual systems in the network. It need not be the only language interpreted by a virtual system since the processors of a system may be quite different.

AMINOL is not a language for physical operating system implementation since such languages should be chosen in a machine dependent way to implement machine dependent physical resource management procedures. The virtual systems factor the responsibilities (and associated authorities) for resource management between the physical operating system (local to a particular network node) and a virtual operating system (global to the entire network). This allows maximum freedom to allow both node implementation managers and virtual process managers to meet their own management goals.

AMINOL does not dictate the virtual operating system procedures, but it does provide a language in which the specification of such procedures

may be safely delegated to the process managers and in which they may in turn safely delegate management responsibilities in a hierarchy of processes. In effect it also plays the role of a network job control language with user defined structures.

1.1 Purpose

This manual is intended to provide a virtual machine programmer with sufficient information to design and implement virtual machine software. No operating systems or application programs are discussed here. Neither do we concern ourselves here with implementation of virtual systems. We will not attempt to explain how or why various language structures were designed, nor will we attempt to demonstrate how to exploit them. These matters are discussed elsewhere.

We will present a formal syntactic specification of process state structures, an informal specification of the processor transformations of process states, and a description of the primitive operators that are currently built into our implementation.

Our implementation of a physical node supporting multiple multi-processor virtual systems is currently written in FORTRAN as a prototype or laboratory system. We choose FORTRAN not for efficiency but rather for implementation portability so that others may readily exploit our work in further

research, development, and applications. They will find our prototype implementation can be used as a normal FORTRAN program in almost any host environment.

Our intent is to produce in this, and subsequent reports, a sufficient set of formal and informal specifications of the entire system to be used as a research laboratory in operating system design and as a pedagogical example of an advanced, comprehensive system design.

1.2 Origins

The virtual system structures and their design principles are described in [KF] and in more detail in a Ph.D. thesis [K]. The resource and process state structures are described in detail in a Ph.D. thesis [C]. The structure of AMINOL has been described previously as part of an extensive set of class notes for an advanced graduate systems course.

The system, process, and resource structures were carefully designed so as to safely delegate the virtual operating system structures to a hierarchy of user defined processes in a manageable way, even if some of the processes are un-cooperative. The expression state structures (activation records) were designed to provide a very high level, macro based, definitionally extendable programming language which is well suited for the implementation of virtual machine software and applications. The language must

be able to manipulate ordinary data as well as processes, resources, expression states, and their interactions, while preserving system integrity.

The expression state structures contain a generalization of the environmental structures found in such languages as PL/1, ALGOL, and SIMULA, and provide very sophisticated control of variable binding, scope, and extent. The concepts of " name " have been significantly generalized and a variety of name binding types are provided.

The macro structure of the program stream owes much to the TRAC language [M] and is essentially a generalization of it to provide a richer name structure and extendable primitive operators.

1.3 Specifications

Although, at various stages of design, a completely formal definition of the AMINO language (AMINOL) and system (AMINOS) has been made, it is not the easiest way to learn AMINOL but is more useful for reference and formal proofs. This manual contains a formal syntactic definition of the system state structures directly manipulated in AMINOL and an informal semantic specification of the processor operations in these state structures. The level of detail and method of presentation is quite similar to that in the ALGOL 60 [N] paper.

1.4 Acknowledgements

This manual reports part of the work of a number of graduate students and reflects many discussions both in and out of advanced system classes. It is not possible to acknowledge each but the authors are sincerely grateful for the many hours of discussions (and arguments) that have led to a deeper insight into the basic system problems attacked in this project.

The contributions of some have been so explicit and extensive that we would like to extend a deep measure of appreciation to and acknowledge the contributions of the following:

R. Brody	R. T. Johnson	W. Bullen
G. Cowan	J. F. Kramer	
J. H. Hine	W. A. Zimmerman	

We would also like to thank The Wisconsin Alumni Research Foundation for a series of research grants for the support of a research assistant and some computer time.

2. OBJECTS AND VALUES

The structures and relationships of objects and values are crucial to the design of any programming language. Many machine languages will define the object structure as a vector of indexable (addressable) locations, each containing a fixed length bit or character string. A high level language commonly introduces arrays and may provide a variety of object structures. Most such languages will only allow operations on values in fixed object structures. The object structures themselves may be declared but not transformed. In such languages the distinction between objects and values is not very important.

In this language we not only provide "high level" objects and values but also provide operators on both structures.

2.1 Syntax

```
<object list>          ::= <object>|<object list><object>
<object>               ::= <simple object>|<structured object>
<structured object>    ::= [<object list>]
<simple object>        ::= [ ]|[<value>]
<value>                ::= <typed value>|<value string>
<typed value>          ::= <type><argument>
<type>                 ::= <value string>
<argument>             ::= <object>
<value string>         ::= <non-meta>|<value string><non-meta>
<non-meta>             ::= <any character except [ and ]>
```

2.2 Object Structures

A clear distinction between objects and values is maintained both in their syntactic and in their semantic structures. An <object> is a nameable location which is either empty, contains other <object>'s , or contains a <value> . A <value> is nameable only by the name of a containing <object> . Operators may reference and transform either or both <object>s and <value>s . In effect, an <object> supplies an address structure for its contained <value>s .

We will introduce a pair of meta characters, (and) , which will be used to represent <object> structure and delimit <value string>s . A <value string> can contain any non-meta characters in the reference language. Any implementation will specify a more restricted set.

2.3 Value Structures

A <value string> is an unstructured character string. We restrict access to the internal structure of a <typed value> to operators designed to operate by the conventions of that type. We may thus introduce arbitrary representational conventions. An implementation may select any desired representation and design the operators accordingly, thus avoiding all unnecessary constraints and inefficiencies.

For the purposes of the reference representation only, we will represent the "value" of a <typed value> as an <argument> . Any reference

language evaluation will be invariant to the representation of each <type> chosen in a given implementation.

2.4 Examples

xy$.5	is a <value string>
(XY$.5)	is a <simple object>
( )	is a null <simple object>
(XY) ($) (.5)	is an <object list>
((ab) (4.))	is a <structured object>
((X) (Y)) ( ) (z)	is an <object list>
(((aaa)) (((b))))	is a <structured object>
XYZ(ABCD)	is a <typed value> of <type> XYZ containing a <value string>
XQT((aB) (Cd))	is a <typed value> of <type> XQT containing an <object list>
X.2(((G.3) ( )))	is a <typed value> of <type> X.2 containing a <structured object>
(ABC((1) (23)))	is a <simple object> containing a <typed value> of <type> ABC, which contains an <object list>

3. ARGUMENT INTERPRETATION

Any <object> used as an <argument> will always be interpreted to contain a <name> . There are four evaluation contexts in which an <object> will be used as an <argument> : in an indirectly named <object> , in a <typed value> , in a <declaration> , or in a <parameter> of a <macro call> . The details of the local environment structures in which the last two cases occur will be discussed later. For now, we can consider a <declaration> to be the association of a <symbol> with an <argument> which it names, and a <parameter vector> to be the current ordered set of <parameter>s of the macro being executed. The local environment will define an ordering on all such contexts.

A <name> will always be interpreted as naming an <object>, an <argument> , or a primitive operator. The naming of an <argument> is interpreted as an indirect reference to whatever is named by the contents of that <argument> in its environment. The naming of an <object> is interpreted as a direct reference to that <object> . The naming of a primitive operator is interpreted as an invocation of that operator in the processor.

We will provide a variety of forms for names with each form representing a different kind of binding (or local environment search) to the named entity.

3.1 Syntax

<declaration> ::= ((<symbol>) <argument>)

<name> ::= <symbolic>|<literal>|<literal><object list>|<typed value>

<symbolic> ::= <primitive name>|<primitive name><address>|<symbol>|<symbol><address>|<parameter name>|<parameter name><address>

<literal> ::= <object>

<primitive name> ::= . .<symbol>

<parameter name> ::= <direct index>

<symbol> ::= <symbol char>|<symbol><symbol char>

<address> ::= <index>|<address><index>

<index> ::= <direct index>|<indirect index>

<direct index> ::= . <nz integer>|. <sign><nz integer>

<indirect index> ::= .

<symbol char> ::= <any <non-meta> except .>

<integer> ::= <numeric string>|<typed integer>

<typed integer> ::= NUM(<signed numeric string>)

<numeric string> ::= <signed numeric string>|<unsigned numeric string>

<signed numeric string> ::= + 0|<sign><nz integer>

<unsigned numeric string> ::= 0|<nz integer>|

<nz integer> ::= <non-zero digit>|<nz integer><digit>

<sign> ::= +|-

<digit> ::= 0|<nz digit>

<nz digit> ::= 1|2|3|4|5|6|7|8|9

3.2 Name Binding

A <primitive name> is bound to a processor action and is analogous to a machine operation code. These names have both scope and extent over all environments in a process. <Primitive name>s may be used only in contexts calling for the execution of their associated actions.

A <literal> name is bound to itself as an <object> and is a generalization of "initial values" (e.g. constants) to "initial objects". <Literal>s may be transformed as any other <object> and may be used in any context where a <name> is required, if an <object list> is used as a <name>, only the first <object> will be considered as a <literal> with the remaining <object>s ignored.

A <parameter name> is bound to the indexed element of the current parameter vector and is a generalization of a "formal parameter name" in an ALGOL procedure. The index value is contained in the <parameter name> itself. In effect, a <parameter name> is an <address> over an implicitly named parameter vector represented as an <object> . A <parameter name> may be used in any context where a <name> is required.

A <symbol> name is bound to an <argument> in a <declaration> with matching <symbol> component and is a generalization of a variable name bound in a structured environment. Each usage of a <symbol> name invokes a symbol lookup in the environment for the matching <declaration> which currently has extent into the local environment. The <symbol> is then interpreted to name whatever is named in the associated <argument> . A <symbol> name may be used in any context where a <name> is required.

A <typed value> name may be interpreted according to the conventions for that <type> (e.g. a <type> "execute only" which would be interpreted by the processor as an <argument> if and only if it were being referenced by the processor for execution as a macro.) Please see the section on primitive operators for a discussion of these <type>s .

3.3 Addressing

An <address> may be appended to any <symbol>, <parameter name>, or <primitive name>. An <address> is a sequence of <index> components which are interpreted from left to right by the following procedure:

a) If leftmost address component is a <direct index> , the indexed sub-object of the current <object> is made the new current <object> .

b) If leftmost address component is an <indirect index> , the <object> named by interpreting the current <object> as an <argument> is made the new current <object> .

c) Delete the leftmost address component and repeat above until address components are exhausted.

The resulting current <object> is the one selected by the <address> . Note that indexing occurs from the left if <index> is positive and from the right if <index> is negative and selects the indexed <object> in the <object list> immediately contained in the current <object> . If a directly indexed current <object> is simple or if the <index> is too large, a fault condition will be returned as described in the next section. The environment to be used in finding an indirect reference is always the local environment of the current <object> .

3.4 Name Faulting

A valid <name> may possibly not be defined in the current environment. In such a case, a fault condition will be generated and interpreted by the processor. These fault conditions will cause actions defined by the operator making the reference. The following fault conditions may arise in the decoding of a <name>:

TABLE 1: NAME BINDING FAULT CONDITIONS

Fault Condition	Cause
NOSYMB	the referencing <symbol> does not match a <declaration> entry in the current local environment
NOARG	a null or non-existent <argument> has been referenced
NOOBJ	an index has specified a non-existent sub-<object> of an <object>
NOPAR	no <parameter> corresponding to the referencing <parameter name> exists in the local environment
INVIDX	improperly formed <address>
PRIM	<primitive name> used other than in the <argument> of the first <parameter> of a <macro call>; or an unimplemented primitive operation has been named
<type>	improper attempt to use a <typed value> as a <name>, or an attempt to index into a <typed value>
INOBJ	reliability fault, <object> has been marked inaccessible as a result of some previous system failure

3.5 Examples

Consider the following three declarations, (in order, X the most recent) to constitute the local environment:

((Y) (((ABC) (D))))

((Z) (Y))

((X) ((() (Y)) () ((Z) (XQT (A)))))

Then we can examine the following <name>'s and the <object>'s they name:

<name>	<object> named, or fault condition
Y	((ABC) (D))
X.+1,X.1,X.-3	(() (Y))
X.+1.-1,X.1.2	(Y)
X.	(() (Y))
X.-1.-1	(XQT (A))
X.-3.2	(Y)
X.-3.2.	((ABC) (D))
X.-3.2..	(ABC)
X.-3.2..2	(D)
X.2,X.+2,X.-2	()
X..-1	(Y)
X..-1..	(ABC)
X.4	NOOBJ
X.1.2.1	NOOBJ
Z	((ABC) (D))
Z.2	(D)
X.3.-2..2	(D)
X.-2.	NOARG
X.A,X.3B	INVIDX
X.-1.-1.1	NOOBJ
X.-1.-1.	XQT
.2	the <object> named by the second <argument> in the current <parameter vector>
.-1	the <object> named by the last <argument> in the current <parameter> vector

4. EXPRESSION STATE STRUCTURES

An expression state is an entity containing information describing the current state of program stream generation and interpretation. In conventional systems the program stream is the sequence of executed instructions and the interpretation of a "program" would correspond to a single expression state. Multiple task or co-routine evaluations would involve multiple expression states. Expression states are represented as <typed value> s, since they may be operated on only by AMINOS sub-processors specifically designed for expression state interpretation. Different sub-processors may operate on different types of expression states but, for the present, we shall restrict our discussion to the GP (general programmable) expression state. Our specification in the next section of the internal structure of a GP type expression state is not a description of an implementation. Implementers may choose another representation to suit their needs without observable effects on any evaluation except for execution efficiencies. An expression state can be considered to consist of two stacks: the generator stack, which describes the current state of program stream generation (corresponding, in conventional systems, to the instruction counter, instruction buffer and program); and the macro stack, which contains the current state of program stream evaluation (corresponding to a program work area or "activation record").

4.1 Syntax

<expression state> ::= GP ⌊<generator stack><macro stack>⌋

<generator stack> ::= <program level>|<generator stack> ⌊<program level>⌋

<program level> ::= <program triple>|<program level> <program triple>

<program triple> ::= ⌊<link><traversal pointer> <program stream buffer>⌋

<link> ::= ⌊<literal>⌋|⌊(<direct address>)⌋|⌊(<direct address>⌋ ⌊<symbolic>⌋)

<direct address> ::= <direct index>|<direct address> <direct index>

<traversal pointer> ::= ⌊ ⌋|⌊<address>⌋

<program stream buffer> ::= ⌊ ⌋|⌊<value string>⌋

<macro stack> ::= ⌊<macro level>⌋|<macro stack>⌊<macro level>⌋

<macro level> ::= <macro call>|<macro level><macro call>

<macro call> ::= ⌊<invocation><parameter vector>⌋

<invocation> ::= ⌊<macro initiation><cut off>⌋

<macro initiation> ::= <null>|(|+|¢

<cut off> ::= <null>|*

<parameter vector> ::= <parameter>|<parameter vector><parameter>

<parameter> ::= <dictionary segment><argument>

<dictionary segment> ::= ⌊ ⌋|⌊<declaration list>⌋

<declaration list> ::= <declaration>|<declaration list><declaration>

<null> ::=

4.2 The Generator Stack

The <generator stack> is a sequence of <program level>s, each level consisting of a sequence of <program triple>s . A <program triple> consists of a <link>, a <traversal pointer>, and a <program stream buffer> . The <link> is an <argument> identifying the <object> to be used for program stream generation. The <name> in the <argument> is either a <literal> or a <direct address> . A <direct address> is implicitly decoded to point directly to the <parameter><argument> or <dictionary segment> <declaration> containing the <link> named <object> . The <traversal pointer> is an <address> identifying the current point of program stream generation. The <program stream buffer> contains the <value string> which is generating the program stream. The last program triple at the last level is active , i.e. it is currently defining program stream generation.

When a <program triple> is initially stacked, the <link> is set to identify the generator object, and the <traversal pointer> and the <program stream buffer> are set to null. The <traversal pointer> is then used to step over the <link> named generator object, left to right, from <simple object> to <simple object> (any <structured object> may be considered to possess a tree structure; since only terminal modes are visited, the form of traversal may be considered as either preorder or postorder).

The contents of each <simple object> in turn is copied into the buffer, and generation proceeds character by character (<typed values> are treated as a single character). When the buffer is exhausted the <traversal pointer> is stepped to the next <simple object> to continue program stream generation. When the program generator object has been exhausted, the corresponding <program triple> is removed, and generation proceeds with the previous <program triple> at the same level. However, the initial <program triple> at a level will not be removed and will continue to return right parentheses, ")", to the program stream. The level will be deleted only when the corresponding <macro level> is popped as described later. If the traversal references to the program generator object return a fault condition, the object is considered exhausted and generation continues as described above.

We have described how the character sequence which forms the program stream is generated. Before we describe how these characters are interpreted, we must define the <macro stack> .

4.3 The Macro Stack

The <macro stack> represents the current state of evaluation of the program stream; this section will define its structure and the following section will describe how it is built and transformed.

The <macro stack> is composed of <macro level>s which correspond, one to one, to the levels in the <generator stack> . Each <macro level> will represent the evaluation of the last <macro call> at the previous level and consists of a sequence of incompletely evaluated <macro call>s . A <macro call> consists of an <invocation> , which will determine the disposal of the macro value, and a <parameter vector> , which will determine the macro <argument>s and their local <dictionary segment> environments.

The last <parameter> of each incomplete <macro call> is an incomplete <parameter> which has not been completely constructed. The last <macro call> at each <macro level> , except the last level, has been completely constructed and is being evaluated at the next level. The last <parameter> of the last <macro call> of the last <macro level> is the current building <parameter> that identifies the current environment and the current building <argument> . Only the <dictionary segment> and the <argument> in the current building <parameter> are active (i.e. currently having their values constructed).

The <macro initiation> of a <macro call> determines the disposition of the value of that macro when it is obtained. The <cut off> of a <macro call> <invocation> , if not null, will prevent extension of the local environment at that point to more global environments.

4.4 Macro Call Building

A <macro call> is built dynamically from successive characters in the program stream or from the results of a just completed macro execution (as described in the next section). Certain characters in the program stream will be treated as control characters whose control actions will cause the decoding of the program stream into <macro call>s. These decoding rules are as follows where X is the next program stream character:

a) If $X \in \{(, +, \phi\}$ then stack a new <macro call> , $\lfloor(X)\rfloor\lfloor\rfloor\lfloor\rfloor\rfloor$, on the last <macro level> .

b) If X = , then add a new <parameter> , $\lfloor\rfloor\lfloor\rfloor$, as the last <parameter> of the last <macro call> of the last <macro level> .

c) If X = ' (a single quote) then fetch the next character and treat it as a non-control character no matter what it is.

d) If X = " (a multiple quote) then all following characters up to a matching right parenthesis will be generated and treated as non-control characters except for counting any included ", (, +, ϕ as left parenthesis in the matching computation. (Single quoted parenthesis are not included in the count.)

e) If X = % then replace the current building argument, Z, with $\lfloor Z\rfloor$.

f) If X is a typed value or a non-control character, it is "concatenated" with the contents of the current building argument as shown in Table 2.

g) If X =) then evaluate the just completed <macro call> as described in the next section.

After application of the appropriate rule above, the next X in the program stream is generated and the rules are applied again.

Table 2: Implicit concatenation rules. The new current building argument is given by the table values. The following abbreviations are used:

VS = <value string>
TV = <typed value>
O = <object>

OLD CURRENT BUILDING ARGUMENT	VALUE BEING RETURNED FOR CONCATENATION				
	VS_2	TV_2	$\lfloor O_2 \rfloor$	$\lfloor O_{2n} \rfloor \cdots \lfloor O_2 \rfloor$
NULL	VS_2	TV_2	$\lfloor O_2 \rfloor$	$\lfloor O_{2n} \rfloor \cdots \lfloor O_2 \rfloor$
VS_1	$VS_1 \; VS_2$	$\lfloor VS_1 \rfloor \lfloor TV_2 \rfloor$	$\lfloor VS_1 \rfloor \lfloor O_2 \rfloor$	$\lfloor VS_1 \rfloor \lfloor O_{2n} \rfloor \cdots \lfloor O_2 \rfloor$
$\lfloor TV_1 \rfloor$	$\lfloor TV_1 \rfloor \lfloor VS_2 \rfloor$	$\lfloor TV_1 \rfloor \lfloor TV_2 \rfloor$	$\lfloor TV_1 \rfloor \lfloor O_2 \rfloor$	$\lfloor TV_1 \rfloor \lfloor O_{2n} \rfloor \cdots \lfloor O_2 \rfloor$
$\lfloor O_1 \rfloor$	$\lfloor O_1 \rfloor \lfloor VS_2 \rfloor$	$\lfloor O_1 \rfloor \lfloor TV_2 \rfloor$	$\lfloor O_1 \rfloor \lfloor O_2 \rfloor$	$\lfloor O_1 \rfloor \lfloor O_{2n} \rfloor \cdots \lfloor O_2 \rfloor$
$\lfloor Im_1 \rfloor \cdots \lfloor Im_n \rfloor \lfloor O_1 \rfloor$	$\lfloor Im_1 \rfloor \cdots \lfloor Im_n \rfloor \lfloor O_1 \rfloor \lfloor VS_2 \rfloor$	$\lfloor Im_1 \rfloor \cdots \lfloor Im_n \rfloor \lfloor O_1 \rfloor \lfloor TV_2 \rfloor$	$\lfloor Im_1 \rfloor \cdots \lfloor Im_n \rfloor \lfloor O_1 \rfloor \lfloor O_2 \rfloor$	$\lfloor Im_1 \rfloor \cdots \lfloor Im_n \rfloor \lfloor O_1 \rfloor \lfloor O_{2n} \rfloor \cdots \lfloor O_2 \rfloor$

Various <typed value>s will be introduced that are designed to bypass or shorten the macro call building procedure above. These will be discussed with the type conversion operator although they may involve automatic untyping by the processor while interpreting the program stream.

4.5 Macro Call Execution

Encountering a right parenthesis in the program stream completes the current building argument and completes the generation of a <macro call> . A <macro call> consists of an <invocation> and a vector of <parameter>s. In a non-dummy <macro call> the <argument> of the first <parameter> is interpreted as the <name> of the macro. The macro name is looked up in the current local environment. If it is a <primitive name> we have a primitive macro call; if it names an <object> we have a non-primitive macro call; if the <initiation> is null, we have a dummy macro call.

If the macro names a primitive operator, that operator is executed to produce its side effects and a value. These operators are described in section 7. The executed <macro call> is then deleted from the current <macro level> and the value (always an <argument>) is returned by the following rules where X is the <macro initiation> of the executed <macro call> and v is the value <argument>:

a) If X = ¢ , v is discarded. This is a null return.

b) If X = (, the contents of v are concatenated onto the current building argument as defined in Table 2. This is a neutral return.

c) If X = + , a new <program triple> , ((P) ( ) ( )) is added to the current <generator level> . This is an active return. P is the <object> named by v .

Program stream evaluation then resumes by the usual rules. Note that X can be null only for a dummy macro call as described below.

If the macro names an <object> , a dummy macro call, (( ) ( ) ( )) , is pushed onto the <macro stack> as a new <macro level> , and a new <program triple> , (((A)) ( ) ( )) where A is the <direct address> of the <object> named by the macro, is pushed onto the <generator stack> as a new <generator level> . If the generator object had been found in an environment global to the expression state, the link will be set to contain the original symbolic macro name as well as the <direct address>: (((A) (<symbolic>)) ( ) ( )) . These actions initiate the evaluation of the non-primitive macro. Program stream evaluation then resumes by the usual rules.

If the <macro call> is a dummy, receiving a right parenthesis completes the evaluation of the last <macro call> at the previous <macro level> . The value to be returned is the <argument> of the last <parameter> of the current <macro level> . After saving this value, both the <macro stack> and the <generator stack> are popped. We then continue as though the last <macro call> at the new <macro level> were a primitive

macro call with its value being the saved <argument> above.

Various <typed value>s will be introduced in the section on primitive operators. Some of these will involve modifications or extensions of the macro execution rules above. One is particularly worth mentioning here, the "execute only" <type> . If an "execute only" <typed value> is encountered in any place other than as a macro name to be executed, it will cause a fault (see next section) to <type> "XQT". If it is used as a macro name it will cause an indirect reference to the <argument> which is represented by the <object> labeled by the <type> . However, no subsequent addressing may be applied to an "execute only" <typed value> .

4.6 The Fault Mechanism

One of the more elegant features of AMINOL is the fault mechanism, enabling abnormal conditions to be handled and/or corrected in the environment in which they occur.

An abnormal event within the processor will generate a fault condition. The fault conditions which may arise during <name> decoding were described in Table 1. In addition, the individual operators may be the source of fault conditions; for example, an arithmetic overflow will generate the OVRFLO condition. Some of these conditions may be handled within the processor itself, and will be transparent to the user.

When the processor itself is not able to handle a fault condition, a fault is generated. Faults will only occur during the interpretation of a <macro call> . Each fault has a unique <name> , and this <name> is looked up in the local environment of the faulting <macro call> . If it is not found, the macro call is treated as a no-op with a null <argument> as its value. If it is found, the macro call is not deleted and is treated as a non-primitive <macro call> , whose macro name is the name of the fault. The <object> named by the fault name is used as a program generator, and a new level is created in both the generator and macro stacks just as for ordinary macro expansion. This level has access to all of the <parameter>s of the faulting <macro call> , and may inspect and/or change them using the primitive operators TSARG and ITSARG . The completion and return of value for the faulted primitive <macro call> occurs just as though it had been a non-primitive <macro call> with the fault name. However, a facility of the JUMP operator allows deletion of the current level and reexecution of the last <macro call> at the previous level. This permits dynamic error checking and correction in the execution environment of the error. (The operators TSARG, ITSARG, and JUMP are fully described in sec. 7).

4.7 Local Environment

Each <argument> , whether part of a <declaration> or a macro call parameter vector, possesses a local environment. This environment contains all the <object>s nameable from the <argument> in question and is defined by ordered sequences of dictionary segments and parameter vectors. Whenever a non-primitive <name> is looked up, the environment local to the <argument> in which it resides is searched.

If the <name> is a <primitive name> it returns a fault condition to the processor. If the primitive name is being looked up for execution by the processor, the fault condition will cause the appropriate primitive operator to be executed.

If the <name> is a <parameter name> , it can reference only the <parameter vector> of the last macro call at the macro level previous to the level containing the reference. The <name> contained in the referenced <argument> is then looked up in its local environment.

If the <name> being looked up is a <symbol> , there are two cases to be considered: if the <symbol> resides in an <argument> in a <declaration> in a dictionary segment, only the previously declared entries in that dictionary segment are searched; if the <symbol> resides in an <argument> of a <parameter vector> the <dictionary segment> associated with the <parameter> containing that <argument> is searched in order, from the most recent to the least recent <declaration> . If a

dictionary segment has been searched unsuccessfully, the previous (in the environment chain) dictionary segment must be searched. Each dictionary segment of a given macro call parameter vector will have the same previous dictionary segment. If the macro call initiation is not null (i.e. not a dummy macro call), the previous dictionary segment is the one associated with the last <parameter> of the previous macro call. If the macro call initiation is null, the link of the first program triple at the current <generator level> contains the address in some <parameter> of the associated program generator <object> , and the local environment of this <object> is considered to be the declaration environment of the macro call. Dictionary segment search will then continue in this declaration environment. Whenever a <symbol name> is found, the <name> contained in the associated <argument> is looked up in its local environment; this process continues until either an <object> is named, or a fault condition is generated.

It is possible that at some point no previous dictionary segment exists. This occurs when the <invocation> of the <macro call> contains a <cutoff> of * . This may happen naturally, when the end of the first dictionary segment in an expression state is reached (and there are no containing expression states), or it may happen as the result of executing the environment cut-off primitive, ECO. This primitive will

mark the environment chain at the point of the previous macro call such that all prior dictionary segments will be rendered inaccessible. The <parameter vector> of the last <macro call> of the previous <macro level> will still be referenceable.

An expression state appears as a <typed value> within an <argument> of a <declaration> . When a dictionary segment search for a <symbol> has exhausted all segments in the environment chain within an expression state (and has not reached an environment cut-off), the search continues in the global environment which is the local environment of the <declaration> containing that expression state. The <parameter vector> previous to the initial <macro level> of an expression state is not referenced and does not exist.

4.8 Examples

We will use a shorthand notation to represent the generator and macro stacks for hand simulation. A <program triple> is represented as a <structured object> with three sub-<object>'s: the first contains a pointer to the generator object, the second contains the traversal pointer (null if the generator is a <simple object>) , and the third contains the buffer with a pointer to the last character generated. All <program triple>s at a given level will be written on one line, subsequent <program level>s will be written on following lines. A <macro level> in the macro stack will be written on the same line as the corresponding <program level> .

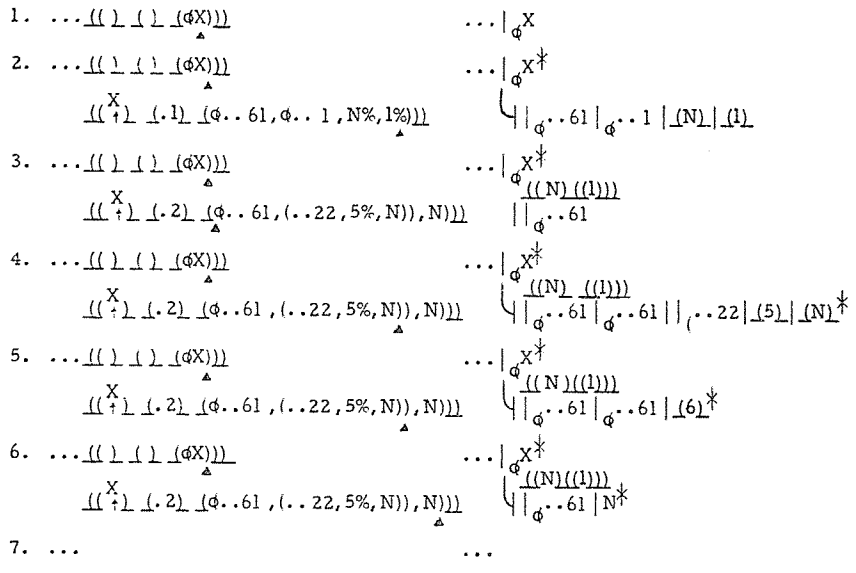
A dummy macro call (i.e. a new <macro level>) will be designated by "ϕ"; the three modes of macro call initiation will be written as "|(", "|+", or "|ϕ"; a comma will be represented by "|"; and a macro call termination (caused by a right parenthesis) will be written as "*". Arguments will appear between the bars representing the call initiations, commas, and terminations; when necessary, the declarations comprising a dictionary segment will be written above the associated argument.

Example 1:

In order to provide a simple example of the usage of this shorthand, we will illustrate the execution of the following generator object named by X, using ..1-DECSYM ,..22-ADD, and ..61-PRINT (send message to printer):

{{X} {{{ϕ..61ϕ..1 ,N%,1%)|(ϕ..61,(..22,5%,N)),N)}}}}

Let X be invoked by ϕX) , then:



Seven phases of the execution have been represented. At phase 1, a macro call initiation has been generated and the first argument, X, has been built. When the macro call, φX) , is completed a level is pushed, with the link identifying the object named by X as the generator object. The traversal pointer begins at .1 , and the buffer is loaded with the simple object named by X.1 . Phase 2 represents the state after the last "%" has been generated, two macro call initiations and four arguments have been built. By phase 3, the DECVAR primitive has been executed, creating a declaration of N in the dictionary segment associated with the first

argument "..61" . The traversal pointer is stepped to .2 as the buffer containing the object named by X.1 has been exhausted. Phase 4 indicates the state after the ADD macro call has been completed. At this point N will be looked up in the local environment, and found in the dictionary segment associated with the last uncompleted argument of the previous macro call. The ADD macro call is deleted, returning the object (6) to be concatenated with the null building argument. In phase 5, the second PRINT call has been completed. The execution of the PRINT will require a message -- (6) -- to be sent to the printer (a foreign system), and will return no value since the invocation was null. At phase 6, the remaining PRINT macro call has been completed, with N as the second argument. However, N does not appear in the local environment of this call, and thus a NOSYMB fault condition will occur on looking up N . We have assumed that there is no declaration for either N or NOSYMB in the local environment, thus the macro call will be treated as a no-op. Phase 7 represents the situation after the exhaustion of the buffer has resulted in the generation of an extra right parenthesis causing the second level in both stacks to be deleted, and deleting the macro call at the (now) current level. Since the generator object at the first level has now been exhausted (we assume), the program triple is also deleted.

Example 2:

There are many forms of name binding available in AMINOL. To illustrate, let the following code be executed for different values of <def> and <ref> , (note: ..1 is DECVAR, ..10 is LITCAL):

```
φ..1 , U%, A% %)
φ..1 , Z%, U% %)
φ..1 , Y%, ¥((Z))%)% %)
φ..1 , X%, ¥(<def>)% %)
φ..1 , Y%, ¥((W))%)% %)
φ..1 , W%, V% %)
φ..1 , Z%, T% %)
<ref>
```

The following sequence of declarations will be created in the dictionary segment of the current building argument:

```
((U) ((A)))
((Z) ((U)))
((Y) ((¥((Z))%)))
((X) ((<def>)))
((Y) ((¥((W))%)))
((W) ((V)))
((Z) ((T)))
```


Then for various replacements for <def> and <ref> , a variety of results may be obtained for the value of <ref> when <ref> is executed:

<ref> \ <def>	Y	(Y)	+Y)	(..10,Y)	+..10,Y)
X	X	X	X	X	X
(X)	Y	((Z))	U	((¥((Z))%))	((Z))
+X)	((W))	T	A	((Z))	T
(..10,X)	(Y)	((Y))	((+Y))	((..10,Y))	((+..10,Y))
+..10,X)	Y	((W))	V	((¥((W))%))	((W))

We will trace the interpretations of three of these references.

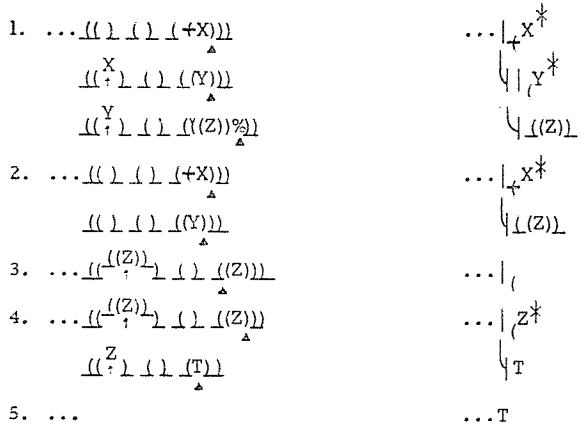
For <ref> = (X) , <def> = (..10,Y)

1. ...((( ) ( ) ((X))) ...|X
2. ...((( ) ( ) ((X))) ...|X*
   (((X↑) ( ) ((..10,Y))) |..10|Y
3. ...((( ) ( ) ((X))) ...|X*
   (((X↑) ( ) ((..10,Y))) |((¥((Z))%))
4. ... ...((¥((Z))%))

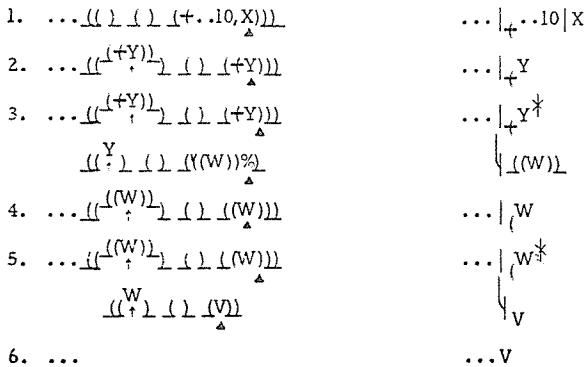
Note that at step 3, the LITCAL macro has been executed, and its value, ((¥((Z))%)) , has been returned to be concatenated with the (null) building argument that was created along with the dummy macro call

invocation. This value is returned to the previous level, when the second level is deleted.

For <ref> = +X) , <def> = (Y):



For <ref> = +..10,X) , and <def> = +Y):



Note that the difference between executing +X) and +..10,X) is that in the former case, a name contained in the generator object named by X (and not declared by code contained in that object), is looked up in the environment local to the declaration of X . In the example given, this means that the declaration of Y after X is not considered, and tne declaration - (Y) (((Z))%))) - is used. In the second example, the generator object, +Y) , has been returned to the current level by the active execution, +..10,X) . Thus the environment in which Y is looked up is not restricted to declarations prior to the declaration of X ; the declaration - (Y) (((W))%))) - is therefore used.

Example 3:

As examples of how iteration and recursion may be expressed in AMINOL, we will consider two versions of a macro which produces the n^{th} Fibonacci number as its value.

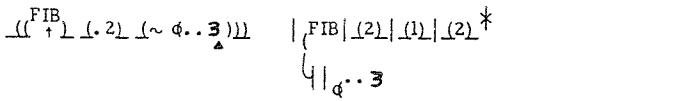
The first uses iteration, stores intermediate values in the parameter vector (via MOVE), and re-executes the original macro call (via JUMP):

```
FIB.1 = (ϕ..3,.2)(..10,.3)(..3,1%,1%))

FIB.2 = (ϕ..6,(..23,.2,1%),.2)
        ϕ..6,(..22,.3,.4)ϕ..6,.4,.3),.4)
        ϕ..3))
```

The value of (FIB, n%,0%,1%) is the n^{th} Fibonacci number (with 0 as the 0^{th}).

An intermediate state of the stacks in the evaluation of (FIB, 4%,0%,1%):



Note how the values ⌊2⌋, ⌊1⌋ , and ⌊2⌋ have replaced the original arguments ⌊4⌋, ⌊0⌋, and ⌊1⌋ respectively.

To implement FIB using recursion:

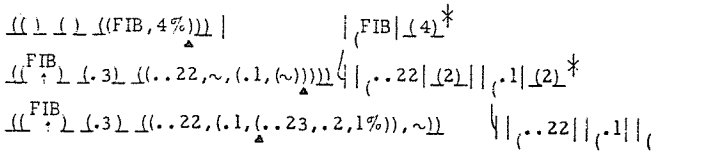
FIB.1 = ⌊¢..3,(..13,.2,+0%))0%¢..3,1%,1%)⌋

FIB.2 = ⌊¢..3,(..13,.2,+1%))1% ¢..3,1%,1%)⌋

FIB.3 = ⌊(..22,(.1,(..23,.2,1%)),(.1,(..23,.2,2%)))⌋

The value of (FIB,n%) will be the n^{th} Fibonacci number, using the recursion formula, FIB(n) = FIB(n-1) + FIB(n-2) with FIB(0) = 0 and FIB(1) = 1 .

An intermediate state in the execution of (FIB,4%):



The "first" recursion, on FIB(3), has been completed, with the value ⌊2⌋ being returned to the second level. The second recursion, on FIB(2), is nearly complete.

5. INTRA-PROCESS INTERACTIONS

We must provide mechanisms to allow both parallel and asynchronous expression state evaluations with controllable interactions between them. The general format of such structures involves multiple processes and systems which will be described later. A special case, contained in a single address space (i.e. a single process state), will be discussed here. One should note that all of the "processes" of conventional single processor systems can be easily represented by a single process in a single system using nested expression states and multiple control and mark points. This is the special case discussed in this section.

5.1 Syntax

The syntactic form of various entities required for non-local interactions are given below. The mutual association of these entities is defined by embedding them in the system and process environment described in section 6.

<ES declaration>::=⌊(<symbol>⌋ ⌊<expression state>⌋)

<control mark point>::=⌊<mark status><ES address><local buffer>⌋

<ES address>::=⌊<symbolic>⌋

<mark status>::=⌊<enable bit><delivery><demand bit>⌋

<local buffer>::=⌊ ⌋ | ⌊<literal>⌋

<control point>::=<system buffer>|<control point><system buffer>

<system buffer>::=<CP buffer name><local buffer>

<CP buffer name>::=<arm bit><symbol>

<arm bit>::=0|1

<enable bit>::=0|1

<demand bit>::=0|1

<delivery>::=<all>|<highest priority>

<all>::=1

<highest priority>::=0

5.2 Nested Expression States

We will first provide for multiple expression states in a single, well ordered environment (address space). We will allow the definition and declaration of different types of expression states. The rules governing the proper nesting of expression states are dependent on the types of the containing expression states. The expression state structures developed in the previous sections all belong to the same type, GP, and the system language described in this report is evaluated by means of the GP sub-processor. We will not discuss other types in this report except to note their possible existence. We will refer to ES when the type of expression state is irrelevant.

An ES may be created in a GP ES by declaration in the dictionary segment of the current building parameter with a local symbol name and a standard initial typed value. The global environment of the new ES is the local environment of its declaration. Further nesting will follow the same rules and we need only discuss them for one level of nesting.

An initial GP ES will be defined such that the initial invariant program generator will generate a parameter vector on the first macro level with no indirection to a global parameter vector. Thus global names must contain a <symbol>. The lookup of symbolic names will continue (unless cut off via the ECO primitive) into the global environment of the ES. If a new generator stack level is to be formed with a globally named generator object, it will be flagged as a global reference.

If two ES's of the same type are nameable in the local environment in a GP ES, then a copy of one may be used to replace the other. The globally bound generator levels of the new ES copy that were bound outside that ES (some globally bound levels could have been bound internally due to internal ES nesting) will be modified on first reference, to bind into the new ES global environment. In effect, such broken bindings will be re-established by symbolic look-up in the new environment as if they were being created for the first time. If the associated macro name now becomes locally bound (due to parameter change or local indirection) the generator level will become locally bound. Although global binding can

become local after a move, local bindings will always remain local.

An ES can be destroyed either by replacement, as described above, or by the removal from the macro stack of the dictionary segment containing it.

5.3 Serial Control

The evaluation of an ES is acomplished by applying a system sub-processor of the same type to that ES. In order to guarantee that the transformations will be well-defined even with potentially overlapping environments, we will constrain all potential sub-processor applications to occur serially, with at most one ES in a nested set of ES's to be transformed at any given time. Before we describe the sub-processor application algorithm we must develop the concepts of mark and control points.

A given ES may or may not have an associated mark point. A given mark point may or may not have an associated control point. Each mark point will have demand or non-demand status and a unique relative priority in an address space (i.e., a nest of ES's). An ES is considered to be a candidate for sub-processor application if and only if

a) there exists a sub-processor of the same type in the local processor

b) there is an associated, demanding, mark point

c) the associated mark point has an associated control point

The candidate with the highest priority associated mark point will have the specified type of sub-processor applied. It will remain applied until it completes one evaluation step in the selected ES. The evaluation step termination is defined by each type of sub-processor. The sub-processor application conditions are tested prior to the next evaluation step and the selected sub-processor (if any) is applied to the selected ES in the next evaluation step. An ES that is having a sub-processor applied will be in active status.

The association of control points with mark points and their transformations will be discussed in the broader context of process states (chapter 6). For the special case discussed in this section, we will assume that the control point-mark point association is invariant and consider only the local transformations of status and ES association.

A mark point may become demanding either upon receipt of a message (as described in the next section) or upon the active ES setting its status to demanding. It may become non-demanding only upon the active ES setting its status to non-demanding.

The mark point associated with the ES can be moved to an unmarked ES of the same type. The local buffer in the mark point may be filled with a message contained in an optional parameter of the transfering macro call. The disposition of the message will be carried out by the destination

ES as described in the next section. Note that a mark point is associated with the declaration containing the ES and copying an ES into another ES declaration does not change either the source or destination mark point association (if any).

5.4 Communications

Messages may be generated by an active ES for transmission to a named control point. Each control point will have a set of unique names that are not in the address space of any ES and are referenced only by the local system processor in delivering messages. In this section we will consider only the special case of intra-ES transmission (i.e. the destination control point is associated with an ES in the same ES nest as the transmitting, active ES).

An active GP ES evaluation step can generate a message and designate the destination by referencing a control point buffer name (in a typed resource). The message will arrive prior to the testing of the sub-processor application conditions for the next evaluation step. The message itself may be any literal.

The receipt of a message is essentially a generalization of the interrupt concept in conventional systems. Each name associated with a destination control point designates a reception buffer in the local system.

Each buffer has an armed or disarmed status. If a message arrives for a disarmed buffer it is simply discarded. If the buffer is armed, the message will replace the current contents (if any) of the buffer. The arrival of messages may affect the status of a mark point and such a change (if any) will occur prior to the sub-processor allocation test for the next evaluation step. This change in status is analogous to an interrupt in a conventional system.

Let M be the set of all mark points such that there is a sub-processor in the local system of the same type as the associated ES, and there is a control point associated with the mark point. Define a subset of M, called M', of mark points that either are demanding or have non-empty local buffers. Let m' be the highest priority member of M'. Define another subset of M, called M'', of mark points that

a) are non-demanding and enabled, and

b) have an empty local buffer, and

c) have at least one non-empty system buffer associated with their control points.

Let m'' be the highest priority mark point in M''. The message to be delivered depends upon the delivery status. The set of system buffers associated with a mark point via the control point are linearly ordered with respect to delivery priority. If the mark point's delivery status is

"highest priority", the contents of the highest priority non-empty system buffer in the set are delivered to the local buffer in the mark point and that system buffer is cleared. If the delivery status is "deliver all", the contents of all non-empty system buffers in the set are concatenated by the argument building rules (in priority order) and converted into a single literal by enclosing the <object list> so obtained in a pair of object brackets. This literal is then delivered to the local buffer and all the system buffers in the set are cleared.

The ES with the higher priority mark point, m' or m" , will be designated to become active on the next evaluation step, and only this mark point may have a message delivered. If the selected mark point is currently not demanding, a message is delivered to its local buffer (if the local buffer is empty), and the demand bit is set. If the mark point is demanding and enabled, a message is delivered if a system buffer is full and the local buffer is empty.

The sub-processor application conditions are then tested as previously described and a new evaluation step is performed. The sub-processor will be applied to the ES which had been designated to become active. If both M' and M" are null sets, no sub-processor will be applied and the system will cycle until some "interrupt" message arrives to demand sub-processor application.

5.5 Examples

Example 1:

Coroutines may be mapped 1-1 onto expression states. In order to provide mutual accessibility in an otherwise linear environment, the "parent" ES will handle the requests for transfer of control.

To initialize, the parent ES is provided with the following object called INIT:

INIT.1 = $\underline{(}$ϕ..30,ES1%,GP%)
ϕ..32,ES1,<msg>)$\underline{)}$

INIT.2 = $\underline{(}$ϕ..38)
ϕ..3,-1%)$\underline{)}$

Then, the call (INIT) will create the first coroutine ES and initialize it. Subsequent returns to the parent ES may then include a message (via ϕ..32,<msg>)). When executed in the parent, these messages may declare new coroutine expression states: ϕ..30,ESn%, GP%) and/or request transfers of control to other coroutine ESs: ϕ..32, ESn,<msg>) .

6. PROCESS STATE STRUCTURES

We have provided, in nested ES's with multiple control and mark points, a very general structure for an address space and expression evaluation. However, all interactions within this address space must occur serially. We will provide disjoint address spaces (process states) whose transformations may proceed in parallel, either synchronously or asynchronously.

6.1 Syntax

<resource set>::=<resource>|<resource><resource set>

<resource>::=[<accountable object declaration>
<primary access declaration>
<secondary access declaration list>]

<accountable object declaration>::=[[<resource name>]
[<marked accountable object>]] |
[[<resource name>]
[<unmarked accountable object>]]

<resource name>::=<symbol>

<marked accountable object>::=[[<marked object list>]<object list>]

<marked object list>::=<marked object>|<marked object list><marked object>

<marked object>::=[<allocable object><mark>] |[<mark>]

<unmarked accountable object>::=[[<allocable object list>]<object list>]

<allocable object list>::=<object list>

<primary access declaration>::=[[<resource name>] [<protected object list>]]

<projected object list>::=XQT[<literal>[<object status>] [<audit trail list>]
<mark status>]

<secondary access declaration list>::= <secondary access declaration>|
<secondary access declaration><secondary access declarat

<secondary access declaration>::=[[<resource name>] [XQT[<literal>]]]

<AO ES>::=AO[<interface buffer><alternate interface buffer>
<communications status>[<resource set>]]

<AO control>::=[<AO control point><AO mark>]

<process state>::=[<AO control>[<AO ES>]]

Undefined syntactic entities have structures of concern only to the AO subprocessor designers and are only indirectly affected by process computations.

6.2 Resources

A <resource> is an address space that can be accessed only by specified macros and whose existence, protection, and retrieval are guaranteed by the system even when the <resource> has been sub-allocated to another, potentially hostile, environment. A <resource> is represented by a <dictionary segment> whose <declaration>s all have the same <symbol> name and their <argument>s all contain "execute only" type values, except the

<argument> in the first <declaration> which contains the accountable object.

The accountable object can be any literal defined by the resource creator or derived from that initial literal via transformations defined by the <primary access declaration>. The interpretation of the accountable object by sub-processor operators depends upon whether or not the resource is of markable type. The meaning of a mark is dependent on the type of resource. For example, we will represent a control point as an allocable object in a control type resource and a mark point as a mark in the control resource.

A markable resource will define meta-messages (commands to the implementation) to be issued when an allocable object is marked or unmarked. This allows specialized conventions to be established between the resource creator and the implementation concerning the interpretation of marked objects. Because of these special requirements, markable resources can only be created with an a priori specified <primary access declarations>. Marks will generally associate with other <declaration>s via a direct address. If a sub-processor eliminates such a <declaration>, it will replace the direct address in the mark with a null value.

The <primary access declaration> will contain (in addition to information required by the system) a <literal> defined by the resource creator whose execution as a macro provides the primary access to the accountable

object. The only access to the accountable object by a GP ES is via the execution of this <literal>. Each <secondary access declaration> contains a <literal> to be executed by a macro call naming the resource. Each secondary access macro is executed in its own environment (i.e. one containing the next inner access declaration of that resource and the outermost access declaration of all other resources in the <resource set>). Each macro may, of course, be provided <parameter>s from the accessing environments. Thus each access macro may use inner access macros as desired and only the primary access macro actually accesses the accountable object. If the accountable object is executed, its environment is only the <resource set> itself.

6.3 Processes

The management of resources in remote environments requires that certain operations (e.g. resource pre-emptions) be reliably carried out without the assistance or cooperation of the local ES. A guaranteed reliable local "agent" to provide such services is built into every process as an AO ES. The structure of an AO ES concerns us only as a "container" for the address space (the <resource set>) of a process state.

Every system in the network contains an AO sub-processor and AO control points will always have a mark, be enabled, and have their buffers armed. The AO mark has the highest priority of any control mark in

the process state. The AO control point and the associated system buffers have names that are unique in the entire network and serve to identify the process state.

Each process state (except for the root process state) has a unique parent process that is responsible for it and has total authority over the resources allocated to it. Each process may create child process states for which it serves as parent. Thus all process states are organized as a process tree with the AO ES enforcing the parental dictates with respect to resources. AO control points provide cooperative, reliable, and secure communication paths for the inter-process management of resources.

The control and mark points introduced in section 5 will be found in markable control type resources in a process. Each mark has a relative priority within a control resource and each control resource has a relative priority within the process. Other resources may contain, as their accountable object, a <literal> whose value is an ES. Thus the entire structure developed through section 5 will fit into a single <resource> of a process. The relative priorities and the mark point set used in the sub-processor application test in section 5 are extended to the entire process address space. At most one sub-processor will thus be applied in any one process step.

The application of a sub-processor to an ES in a process will initiate a process step that terminates in a new process state when the ES

evaluation step completes. Each process step is finite in both real and virtual space and time. The definition of a process step is dependent on the type of ES involved. If the step involves a GPES, the process step will terminate with the execution of the actions associated with a right parenthesis control character in the program stream.

If two processes are in the same virtual system, they will start and finish their process steps at the same (virtual) time and thus proceed in a synchronized parallel manner. The sub-processor application test is always made locally in each process.

Each virtual system goes through a system cycle of three "system steps" (called phases 1, 2, and 3). Phase 1 selects the next sub-processor to be applied to each process in the system and delivers the required messages (if any) to the process interface buffer. Phase 2 applies the sub-processors to dispose of the interface buffer contents and carry out a process step. The process step will complete with any outgoing message in the process interface buffer. Phase 3 will dispatch such messages to the specified system buffers. This may involve intra-or inter-system communication. If the message is intra-system, it will "arrive" and be considered in the next phase 1 of the system.

The distribution of processes amoung the virtual systems is independent of the process tree relationship and may be changed dynamically by processes themselves.

If two processes are in different virtual systems, they will start and finish their process steps in synchronization with each of their local system phases, and thus proceed in an asynchronous parallel manner. Although all virtual systems will proceed through their local system phases with a finite real time duration for each phase, their relative rates are virtually unspecified and unconstrained. Similarly, inter-system messages have a finite but virtually unspecified and unconstrained transit delay before receipt by the destination system buffer.

There are two inter-system transmission constraints. First, all messages transmitted by a system in its phase 3 to another system will arrive at that system at the same virtual time. Second, a subsequent message transmission by a system to another will not arrive before a prior message sent by that system.

Synchronization of asynchronous parallel processes can only be acomplished by normal synchronizing message exchanges initiated by the processes themselves.

6.4 Foreign Processes

The network of virtual systems and the process tree are all subject to constraints sufficient to guarantee the process and resource integrity. However, a general network should allow interactions with unconstrained

systems representing specialized systems such as input-output processors, other networks, or special-purpose processors.

Since the network designers are not controlling the structures of such foreign systems, they must instead specify the interface. When a foreign system is created, it can be given a set of control point names to which it may transmit messages in a specified language. The foreign system will supply a set of destination names to which network processes may transmit messages. The actions taken by a specific foreign system are locally defined by that foreign system.

Since the network designers do not control a foreign system's internal operations, the foreign system processes are unconstrained except for ordinary messages to the network processes. However, the network can not be responsible for processes or resources transmitted to foreign systems and such transmissions are forbidden.

6.5 Typed Resources

The creation and destruction of a <resource> may require network modification. Such a <resource> will be "typed" by the inclusion of a type name in the <object status> of the <primary access declaration>, thus limiting its creation and destruction to special-purpose operators and preventing counterfeiting.

Currently two kinds of typed resources are defined. The "CONTROL" type has been discussed above. The "RECEIVER" type will identify control point buffers to which the process may transmit messages, and will contain in its <accountable object> one or more buffer names. There are four classes of buffers involved, each placing different restrictions on the form of the message which may be sent. The name itself (which is implementation dependent) will serve to identify its class.

1) AO control point buffers - to which secure messages may be sent by the ALLOCATE OBJECT, RETURN OBJECT, DESTROY PROCESS, and PRE-EMPT OBJECT primitives;
2) PR control point buffers - to which secure messages (defining processes and their contained resources) may be sent by the CREATE PROCESS, and MOVE PROCESS primitives;
3) foreign system control point buffers - to which messages (not containing resource objects) may be sent by the TRANSMIT MESSAGE primitive;

4) other (intra-process tree) control point buffers - to which messages (not containing resource objects) may be sent by the TRANSMIT MESSAGE primitive.

A process may have any number of RECEIVER type resources, but it must have at least one: containing those buffer names corresponding to class one. These name the AO buffers in the parent and child processes, and this resources accountable object may be updated only by the CREATE PROCESS and DESTROY PROCESS primitive.

7. PRIMITIVE OPERATORS

MACRO: DECLARE SYMBOL-DECSYM

CLASS: Intra-process non-applicative primitive

ARG 1: ..1

ARG 2: ⌊(<symbol>)⌋

ARG 3: <object>

DESCRIPTION: A <declaration> of the form ⌊(<symbol>)⌋ <object>)⌋ is added to the <dictionary segment> associated with the current building parameter.

VALUE: ARG 2 itself

SIDE EFFECTS: <Declaration> creation in the local environment.

EXAMPLES:

Program Stream	Side Effect	Value
(..1, X% Y%)	see NOTE 1	⌊(X)⌋
(..1, A%, AB% %)	see NOTE 2	⌊(A)⌋
(..1, W%, A%B%)	see NOTE 3	⌊(W)⌋
(..1, Z%, 1%23%%)	see NOTE 4	⌊(Z)⌋

NOTE 1: The <declaration> ⌊(X) ⌊(Y)⌋ is added.

NOTE 2: The <declaration> ⌊(A) ⌊(AB)⌋⌋ is added.

NOTE 3: The <declaration> ⌊(A) ⌊(A) ⌊(B)⌋⌋ is added.

NOTE 4: The <declaration> ⌊(Z) ⌊(⌊(1)⌋ ⌊(2)⌋ ⌊(3)⌋⌋⌋

MACRO: TYPE CONVERSION-TYPE

CLASS: Intra-process non-applicative primitive

ARG 1: ..2

ARG 2: <object>

ARG 3: ⌊(<symbolic>)⌋ OPTIONAL

DESCRIPTION: If ARG 3 is present, it must denote a <type> which is known to the sub-processor. In general, ARG 2 will then be converted to an "implementation-efficient" representation. If ARG 3 is absent, the ARG 2 <object> will be untyped if allowable under the conventions of the <type> involved. The following <type>s are currently handled by the TYPE operator:

NUM - converts a <numeric string> to a <typed integer>. A <typed integer> is a form of <integer> and thus may be used anyplace an <integer> <argument> is required by a primitive operator. NUM may be untyped.

XQT - designates the contents of an <object> to be "execute only". Any reference to an XQT <typed value>, other than as a macro name, will produce an XQT fault. XQT may not be untyped.

ARG - converts the contents of an <object> into the form of a macro call <argument>. When encountered in the program stream, a new <parameter> will be added to the macro stack, and the <argument> of the ARG <typed value> will be inserted as the current building argument. ARG may be untyped.

DICSEG - converts the contents of an <object> into a <dictionary segment>. When encountered in the program stream, a copy of the <typed value> <argument> will be concatenated with the <dictionary segment> associated with the current building parameter. An <object> to be converted must be either null (generating an empty <dictionary segment>) or contain a series of pairs: ⌊(<symbol$_1$>)⌋ <object$_1$>

⋮

⌊(<symbol$_n$>)⌋ <object$_n$>)⌋

A <dictionary segment> of the form ⌊(⌊(<symbol$_1$>⌋<object$_1$>⌋
⋮
⌊(<symbol$_n$>⌋ <object$_n$>⌋)⌋
will be created as the <typed value> . DICSEG may be untyped.

PAR - converts a <structured object> of the form
⌊(DICSEG⌊(~)⌋ ⌊ARG⌊(~)⌋)⌋
or ⌊(DICSEG⌊(~)⌋ <object>⌋
into a <parameter> with the corresponding <dictionary segment> and <argument> . When encountered in the program stream, it will cause a matching <parameter> to be added to the macro stack, becoming the new current building parameter. PAR may be untyped.

MCL - converts a <structured object> into a sequence of <macro call>s . When encountered in the program stream, the corresponding sequence of <macro call>s will be added to the macro stack, the last <parameter> becoming the new current building parameter. The first sub-object of the <object> to be converted must be a <simple object> containing a valid non-null <macro initiation>: ⌊(⌋ , ⌊ϕ⌋ , or ⌊+⌋ . The remaining sub-objects may be any sequence of ⌊PAR⌊(~)⌋s , ⌊ARG⌊(~)⌋s , ⌊MCL⌊(~)⌋s , or <object>s . PAR <typed value>s will be mapped into <parameter>s , ARG <typed value>s will be mapped into <parameter>s with null <dictionary segment>s , other MCL <typed value>s will be mapped into <macro call>s , and other <object>s will be converted to be the <argument>s of <parameter>s . If only a single sub-object (the <macro initiation>) is present, one null <parameter> will be included in the <macro call> . MCL may be untyped.

VALUE: ARG 2 itself

FAULTS: BADARG if ARG 2 cannot be typed or untyped, or if the ARG 3 <type> is not known.

SIDE EFFECTS: The ARG 2 <object> is transformed.

MACRO: JUMP

CLASS: Intra-process control primitive

ARG 1: ..3

ARG 2: ⌊(<integer>⌋ OPTIONAL

ARG 3: ⌊(<integer>⌋ OPTIONAL

VALUE: ARG 2 itself

FAULTS: BADARG if ARG 2 or ARG 3 are of improper form (see below)

SIDE EFFECTS: Modifies the current program triple as specified by the values of ARG 2 and ARG 3.

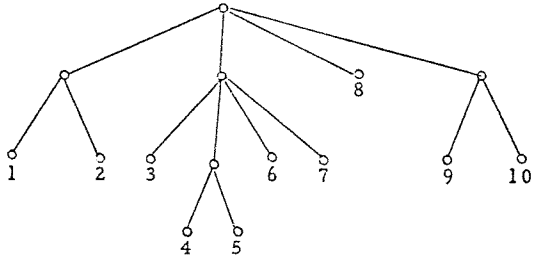
Value of ARG 2	Value of ARG3	Action on Program Triple
not present	not present	Pop an instruction level and associated macro level, reexecute last macro call at previous level.
0	not present	No change
<0	not present	Restart buffer
>0	not present	Skip to end of buffer
0	n	Up n levels (reload) and restart object.
m	n	Up n levels, skip to mth object to right at that level
-m	n	Up n levels, skip to mth object to left at that level.

An attempt to skip to the left past the first node at a level will result in a skip to that first node.

An attempt to skip to the right past the last node at a level will exhaust that level.

An attempt to jump up past the first level will exhaust the object if ARG 2 > 0 and restart if ARG 2 $\leq$ 0 .

EXAMPLE: Assume the following structured program generator object:



Included in program stream at node 5	Action
(..3,0%)	No change
(..3,0%,0%)	Reload and restart node 5
(..3,-1%)	Restart node 5
(..3,1%)	Skip to node 6
(..3,3%)	Skip to node 6
(..3,2%,1%)	Skip to node 7
(..3,0%,2%)	Skip to node 3
(..3,-1%,2%)	Skip to node 1
(..3,-1%,-2%)	BADARG fault
(..3,2%,2%)	Skip to node 9
(..3,10%,1%)	Skip to node 8
(..3,-12%,2%)	Skip to node 1
(..3,1%,6%)	Exhaust program generator
(..3,-1%,6%)	Skip to node 1

MACRO: ENVIRONMENT CUT-OFF - ECO

CLASS: Intra-process non-applicative primitive

ARG 1: ..4

DESCRIPTION: Marks the local environment decoding chain at the point of the previous macro call such that all prior <declaration>s become inaccessible.

The <parameter> vector will remain accessible, however, enabling <argument>s at the previous level to be looked up in their local environment.

VALUE: The null value string

FAULTS: None

SIDE EFFECTS: Environment chain modification

EXAMPLE: <..6,(..4)X,Y) will assure a NOSYMB fault on looking up X, as prior <declaration>s are inaccessible.

In the case of <..6,(..4).1,Y), the execution of the ECO will not affect the binding of .1.

MACRO: NO-OPERATION - NOP

CLASS: Intra-process applicative primitive

ARG 1: ..5

ARG 2: <object>

DESCRIPTION: ARG 2 is simply referenced. This operator thus allows fault testing of the operands.

VALUE: ARG 2 itself

FAULTS: None

SIDE EFFECTS: None

EXAMPLES:

Assuming the following <declaration>s have extent:

```
((BRANCH) (( ))
((X) (XQT( ))
((Y) (BRANCH.1( ))
((Z) ((A)))
```

PROGRAM STREAM	VALUE
(..5 ,X)	XQT fault
(..5 ,Y)	BRANCH.1 fault
(..5 ,X%)	(X)
(..5 ,Z.2)	NOOBJ fault

MACRO: MOVE

CLASS: Intra-process non-applicative primitive

ARG 1: ..6

ARG 2: <object>

ARG 3: <object>

ARG 4: <integer> OPTIONAL

DESCRIPTION: If ARG 4 is absent, the <object> named by ARG 2 replaces the <object> named by ARG 3.

If ARG 4 is present, it will reference the nth (or - nth) character/sub-<object> of the <simple object>/<structured object> named by ARG 3. In this case ARG 2 and ARG 3 must be both simple or both structured. The contents of ARG 2 (either a <value string> or <object list>) will be inserted to the left or right (as ARG 4 is positive or negative) of the character/sub-<object> designated by ARG 4.

VALUE: ARG 2 itself

FAULTS: BADARG if ARG 4 designates a non-existent character/sub-<object> , or if ARG 2 and ARG 3 are not both simple or both structured.

BADARG if ARG 4 is present and either ARG 2 or ARG 3 names a <simple object> containing a <typed value>.

SIDE EFFECTS: An <object> or <argument> referenced by ARG 3.

EXAMPLES: Assume the following <declaration>s in the current environment:

```
((X) (((1) (2) (3))))
((Y) ( ))
((Z) ((ABCD)))
```

PROGRAM STREAM	SIDE EFFECT	VALUE
(..6,10%,X)	see NOTE 1	(10)
(..6,X.2,X.1)	see NOTE 2	X.2
(..6,X,X.1)	see NOTE 3	X
(..6,12%%,X,1%)	see NOTE 4	((12))
(..6,A%B%,X,-1%)	see NOTE 5	((A) (B))
(..6,4%56%,X,2%)	see NOTE 6	((4) (5) (6))
(..6,4%,X,3%)		BADARG fault
(..6,4%%X,6%)		BADARG fault
(..6,1%,Y)		NOARG fault
(..6,43%,Z,-2%)	see NOTE 7	(43)
(..6,A%%Z)	see NOTE 8	((A))
(..6,A%%,Z,-1%)		BADARG fault

The <declaration>s will be transformed as follows:

NOTE 1: ((X) ((10)))

NOTE 2: ((X) (((2) (2) (3))))

NOTE 3: ((X) ((((1) (2) (3)) (2) (3))))

NOTE 4: ((X) (((12) (1) (2) (3))))

NOTE 5: ((X) (((1) (2) (3) (A) (B))))

NOTE 6: ((X) (((1) (4) (5) (6) (2) (3))))

NOTE 7: ((Z) ((ABC43D)))

NOTE 8: ((Z) (((A))))

MACRO: REMOVE - REM

CLASS: Intra-process non-applicative primitive

ARG 1: ..7

ARG 2: <object>

ARG 3: <integer> OPTIONAL

ARG 4: <integer> OPTIONAL

DESCRIPTION: If ARG 3 and ARG 4 are both absent, the contents (<value> or <object list>) of the <object> named by ARG 2 is removed and becomes the value returned.

If only ARG 4 is absent, ARG 3 will designate the nth (-nth) character/sub-<object> of the <simple object>/<structured object> named by ARG 2 to be removed and returned.

If ARG 4 is present, |m| characters/sub-<object>'s to the left/right (as m is positive/negative) of (and including) the nth or -nth character/sub-<object> of the <simple object>/<structured object> named by ARG 2 will be removed and returned.

VALUE: The removed <value> or <object list>

FAULTS: BADARG if ARG 4 and/or ARG 3 are out of range.

SIDE EFFECT: On <object> named by ARG 2.

EXAMPLES:

Assume the following <declaration>s in the current environment:

((X) (((1) (2) (3))))
((Y) ((ABCD)))

PROGRAM STREAM	SIDE EFFECT	VALUE
(..7,X)	see NOTE 1	(1) (2) (3)
(..7,X.2)	see NOTE 2	2
(..7,X,-1%)	see NOTE 3	(3)
(..7,Y,-3% ,2%)	see NOTE 4	AB
(..7,X,1% ,-2%)	see NOTE 5	(1) (2)
(..7,Y,-1%,5%)		BADARG fault
(..7,X,6%)		BADARG fault
(..7,Y,4%)	see NOTE 6	D

The <declaration>s are transformed as follows:

NOTE 1: ((X) (( )))
NOTE 2: ((X) (((1) ( ) (3))))
NOTE 3: ((X) (((1) (2))))
NOTE 4: ((Y) ((CD)))
NOTE 5: ((X) (((3))))
NOTE 6: ((Y) ((ABC)))

MACRO: TRANSFER ARGUMENT - TSARG

CLASS: Intra-process applicative primitive

ARG 1: ..8

ARG 2: (<integer>)

DESCRIPTION: ARG 2 specifies an index into the current <parameter> vector. For ARG 2 positive, indexing is from the left; for ARG 2 negative, indexing is from the right. A copy of the <argument> associated with the indexed <parameter> is returned.

VALUE: The copied argument

FAULTS: BADARG if ARG 2 is out of range

SIDE EFFECTS: None

Examples: Assume the non-primitive macro call (M,X,Y.2,3%). Then let the following program streams be contained in the generator object associated with M:

PROGRAM STREAM	VALUE
(..8,1%)	M
(..8,-2%)%	(Y.2)
(..8,4%)	(3)
(..8,5%)	BADARG fault

MACRO: INVERSE TRANSFER ARGUMENT - ITSARG

CLASS: Intra-process non-applicative primitive

ARG 1: ..9

ARG 2: <object>

ARG 3: [<integer>]

DESCRIPTION: ARG 2 is treated as an <argument>, the contents of which replaces the <argument> associated with the <parameter> indexed by ARG 3, (see TSARG above).

VALUE: ARG 2 itself

FAULTS: BADARG for ARG 3 out of range

SIDE EFFECTS: Argument vector manipulation

EXAMPLES:

PROGRAM STREAM	VALUE	
(..9,X%,2%)	[X]	see NOTE 1
(..9,Y,3%)	Y	see NOTE 2
(..9,Z.3,-1%)	Z.3	see NOTE 3

NOTE 1: The <value string> X replaces the <argument> currently referencable by .2.

NOTE 2: Assume the following <declaration> of Y:

[(Y) [((ABC) [DEF)]]]

Then the <structured object> [(ABC) [DEF)] replaces the <argument> currently referencable by .3.

NOTE 3: Assume the following <declaration> of Z:

[(Z) [((A) [B) [(C) [D)]]]]

Then the <object list> [C) [D) replaces <the argument> currently referencable by .-1.

MACRO: LITERAL CALL - LITCAL

CLASS: Intra-process applicative primitive

ARG 1: ..10

ARG 2: <object>

ARG 3: [<address>] OPTIONAL

DESCRIPTION: The <literal> named by ARG 2 is addressed by the <address> in ARG 3 to designate a sub-object. If no ARG 3 is provided, the <literal> named by ARG 2 is designated.

VALUE: The designated <literal>.

FAULTS: None

SIDE EFFECTS: None

EXAMPLES: Assuming the following <declaration>s:

[(X) [((1) [2))]]
[(Y) [((ABC) [NUM [+3)]]]]

PROGRAM STREAM	VALUE
(..10,X%)	[X]
(..10,X%,.1%)	NOOBJ fault
(..10,X)	[(1) [2)]
(..10,X,.2%)	[2]
(..10,Y,.(..11,X.-1)%)	[NUM [+3)]

MACRO: VALUE CALL - VALCAL

CLASS: Intra-process applicative primitive

ARG 1: ..11

ARG 2: <object>

ARG 3: ⌊<address>⌋ OPTIONAL

DESCRIPTION: Returns the contents of the <object> named by ARG 2 (as addressed optionally by ARG 3).

VALUE: The designated <value string> <typed value>, or <object list>.

FAULTS: None

SIDE EFFECTS: None

EXAMPLES: Assume the following <declaration>s:

⌊(X⌋ ⌊(⌊(ABC⌋ ⌊(⌊(DEF⌋ ⌊(GHI⌋⌋⌋⌋⌋
⌊(Z⌋ ⌊(⌊(NUM⌋⌊+5⌋⌋⌋⌋

PROGRAM STREAM	VALUE
(..11,ABC%)	ABC
(..11,ABC%,..1%)	NOOBJ fault
(..11,X)	⌊(ABC⌋ ⌊(⌊(DEF⌋ ⌊(GHI⌋⌋
(..11,X,.2.1%)	DEF
(..11,X.1)	ABC
(..11,X,.2%)	⌊(DEF⌋ ⌊(GHI⌋
(..11,X.2.2)	GHI
(..11,Z)	NUM⌊+5⌋

MACRO: LENGTH

CLASS: Intra-process applicative primitive

ARG 1: ..12

ARG 2 <object>

DESCRIPTION: To determine length of value strings contained in simple object and the number of objects in the referenced level of a structured object.

VALUE: A signed numeric string ≥ + 0 .

FAULTS: None

SIDE EFFECTS: None

EXAMPLES: Assuming the following <declaration>s of X and Y .

⌊(X⌋ ⌊(⌊(⌊(ONE⌋ ⌊(TWO⌋ ⌊(THREE⌋⌋⌋⌋
⌊(Y⌋ ⌊(⌊(⌊(⌋ ⌊(⌋⌋ ⌊(⌊(1⌋ ⌊(2⌋ ⌊(⌊(3a⌋ ⌊(3b⌋⌋⌋⌋⌋⌋

PROGRAM STREAM	VALUE
(..12,ABCD%)	⌊+4⌋
(..12,X)	⌊+3⌋
(..12,Y.2)	⌊+3⌋
(..12,Y.1.2)	⌊+0⌋

MACRO: MATCH STRING - MATCH

CLASS: Intra-process applicative primitive

ARG 1: ..13

ARG 2: <simple object>

ARG 3: <simple object>

DESCRIPTION: The values in ARG 2 and ARG 3 are compared. If one value is typed the other must be the same type or no match will occur. If both are of the same type but are not recognized by this primitive operator (it does not know how to compare them) the corresponding type fault will be generated. If the type is recognized, the typed values will be matched.

VALUE: +0 if matched values and +1 if not matched values.

FAULTS: None

SIDE EFFECTS: None

EXAMPLE: Assuming the following definitions of X, Y and Z

VARIABLE	NAMED OBJECT
X	(NUM (+123))
Y	(NUM (+123))
Z	(NUM (+11))

PROGRAM STREAM	VALUE
(..13,ABC%,ABD%)	(+1)
(..13,ABC%,ABC%)	(+0)
(..13,X,+123%)	(+1)
(..13,X,Z)	(+1)
(..13,X,Y)	(+0)

MACRO: SIGN

CLASS: Intra-process applicative primitive

ARG 1: ..20

ARG 2: (<integer>)

DESCRIPTION: To determine whether ARG 2 is less than, equal to, or greater than zero.

VALUE: The <signed numeric strings> -1, 0, or +1 will be returned as the value of ARG 2 is less than, equal to, or greater than zero.

FAULTS: None

SIDE EFFECTS: None

EXAMPLES: Assuming the following <declaration>s of X and Y:

((X) ((NUM (+12))))
((Y) ((-14)))
((Z) (((1) (2))))

PROGRAM STREAM

(..20,0%)	(+0)
(..20,(..23,(..2,5%,NUM%),3%))	(+1)
(..20,(..22,X,Y))	(-1)
(..20,Z)	BADARG fault

MACRO: ABSOLUTE VALUE - ABSVAL

CLASS: Intra-process applicative primitive

ARG 1: ..21

ARG 2: [<integer>]

DESCRIPTION: Forms absolute value of the <integer>.

VALUE: The absolute value of the <integer> as <typed integer> or <signed numeric string> depending on whether ARG 2 was a typed value or a numeric string.

FAULTS: None

SIDE EFFECTS: None

EXAMPLES: Assuming the following <declaration>s of X and Y:

[(X) [(NUM [-37]))]
[(Y) [(NUM [+34]))]

PROGRAM STREAM	VALUE
(..21,-5%)	[+5]
(..21,X)	[NUM [+37]]
(..21,(..22 X,Y))	[NUM [+3]]

MACRO: ADDITION - ADD

CLASS: Intra-process applicative primitive

ARG 1: ..22

ARG 2: [<integer>]

ARG 3: [<integer>]

DESCRIPTION: The two <integer> operands are added exactly.

VALUE: <typed integer> if mixed mode or both arguments typed.
<signed numeric string> if <value string> arguments.
The value returned is the sum of the operands.

FAULTS: OVRFLO if result exceeds implementation precision.
BADARG if a <numeric string> argument is too large.

SIDE EFFECTS: None

EXAMPLES: Assuming the following <declaration>s of X and Y:

[(X) [(NUM [+24]))]
[(Y) [(NUM [-24]))]

PROGRAM STREAM	VALUE
(..22,+3%,-1%)	[+2]
(..22,256%,-0%)	BADARG fault
(..22,X,1%)	[NUM [+25]]
(..22,X,Y)	[NUM [+0]]

MACRO: SUBTRACT - SUB

CLASS: Intra-process applicative primitive

ARG 1: ..23

ARG 2: [<integer>]

ARG 3: [<integer>]

DESCRIPTION: The <integer> operand ARG 3 is subtracted from the <integer> operand ARG 2.

VALUE: <typed integer> if mixed mode or both arguments typed.
<signed numeric string> if <value string> arguments.
The value returned is the difference of the operands.

FAULTS: OVRFLO if result exceeds implementation precision.
BADARG if <numeric string> argument is too large.

SIDE EFFECTS: None

EXAMPLES: Assuming the following definitions of X and Y:

[(X) [(NUM [+24]))]
[(Y) [(NUM [-24]))]

PROGRAM STREAM	VALUE
(..23,+3%,-1%)	[+4]
(..23,256%,-0%)	BADARG fault
(..23,X,1%)	[NUM [+23]]
(..23,Y,X)	[NUM [-48]]

MACRO: MULTIPLY - MUL

CLASS: Intra-process applicative primitive

ARG 1: ..24

ARG 2: [<integer>]

ARG 3: [<integer>]

DESCRIPTION: The two <integer> operands are multiplied exactly.

VALUE: <typed integer> if mixed mode or both arguments typed.
<signed numeric string> if <value string> arguments.
The value returned is the product of the operands.

FAULTS: OVRFLO if result exceeds implementation precision.
BADARG if <numeric string> argument is too large.

SIDE EFFECTS: None

EXAMPLES: Assuming the following <declaration>s of X and Y:

[(X) [(NUM [+6]))]
[(Y) [(NUM [-3]))]

PROGRAM STREAM	VALUE
(..24,+3%,-2%)	[-6]
(..24,256%,-0%)	BADARG fault
(..24,2%,0%)	[+0]
(..24,X,5%)	[NUM [+30]]
(..24,X,Y)	[NUM [-18]]

MACRO: INTEGER DIVISION (with Remainder) - DIV

CLASS: Intra-process applicative primitive

ARG 1: ..25

ARG 2: [<integer>]

ARG 3: [<integer>]

DESCRIPTION: The <integer> operand ARG 2 is divided by the <integer> operand ARG 3 by integer division with remainder.

VALUE: <typed integer> if mixed mode or both arguments typed. <signed numeric string> if <value string> arguments. The value returned is a structured object containing the <integer> quotient and the <integer> remainder of th the division.

FAULTS: OVRFLO on division by zero. BADARG if <numeric string> argument is too large.

SIDE EFFECTS: None

EXAMPLES: Assuming the following <declarations>s of X and Y:

((X) ((NUM (+12))))
((Y) ((NUM (-7))))

PROGRAM STREAM	VALUE
(..25,+7%,+2%)	(((+3) (+1)))
(..25,+5%,-0%)	BADARG fault
(..25,+5%,0%)	OVRFLO fault
(..25,X,3%)	(((NUM (+4)) (NUM (+0))))
(..25,X,Y)	(((NUM (-1)) (NUM (+5))))

MACRO: DECLARE EXPRESSION STATE - DECEXP

CLASS: Intra-process expression state primitive

ARG 1: ..30

ARG 2: [<symbol>]

ARG 3: [<type>]

ARG 4: <object> OPTIONAL

DESCRIPTION: Adds a <declaration> containing an expression state (of ARG 3 <type>) to the <dictionary segment> of the current building parameter: ((<symbol>) (<type> (~))) . The <type> must be known to the sub-processor, and the <object> named by ARG 4 will be interpreted, according to the ARG 3 <type>, as a representation of the initial state of the expression state. Currently, only the GP type is handled, for which ARG 4 must not be present. The initial state of a GP expression state will consist of a single generator and macro level, with the following string in the program stream buffer:
ϕ(..11,(..2,(..38)%,XQT%)))ϕ..33,,,0%)ϕ..15,-1%)
When executed, this will read a message, make it an execute-only type <object>, and execute it. When this first level is returned to, the mark status will be set non-demanding. Then, if subsequently control is given to the ES, the buffer will be re-executed.

VALUE: ARG 2 itself

SIDE EFFECTS: A <declaration> is added to the current environment.

FAULTS: BADARG if ARG 3 names an unknown <type> of ES, or if ARG 4 is of improper form for that <type>.

EXAMPLES: (..30,SAM%,GP%) will add the <declaration>:
((SAM) (GP (~)))
which contains a GP expression state with the initial form as described above.

Note that MOVCM will fault on referencing ARG 2 if not generated from within the <primary access declaration> of a control resource.

VALUE: ARG 2 itself

SIDE EFFECTS: On mark point attachment

FAULTS: BADARG if ARG 2 does not name a control resource, if source ES is not marked, if destination ES is marked, if arguments do not conform to one of the four variants.

MACRO: SET CONTROL MARK STATUS - SETCMS

CLASS: Intra-process control primitive

ARG 1: ..33

ARG 2: [<integer>] OPTIONAL

ARG 3: [<integer>] OPTIONAL

ARG 4: [<integer>] OPTIONAL

ARG 5: <symbol> OPTIONAL

DESCRIPTION: The status of the control mark point associated with the ES named by ARG 5 (or the currently active ES, if ARG 5 is absent) will be modified. Each of the <enable bit>, <delivery>, and <demand bit> will be set; if 0, the bit will be reset; if absent the bit will be unchanged.

VALUE: ARG 2 itself

SIDE EFFECTS: On markpoint status

FAULTS: BADARG if ARG 2, ARG 3, and ARG 4 are not 0, 1, or absent; or if no mark point is attached to the named ES; or if ARG 5 does not name an ES.

EXAMPLES: ϕ..33,,,1%,SAM) will set the <demand bit> of ES SAM, other status will be unchanged.
ϕ..33,0%,0%,,) will reset the <enable bit> and <delivery> of the active ES, the <demand bit> is unaffected.

MACRO: READ CONTROL MARK STATUS - RDCMS

CLASS: Intra-process control primitive

ARG 1: ..34

ARG 2: <symbol> OPTIONAL

DESCRIPTION: The status of the control mark point associated with the ES named by ARG 2 (or the currently active ES if ARG 2 is absent) is returned as a string of three bits: <enable bit><delivery><demand bit>.

VALUE: The <value string> as described.

SIDE EFFECTS: None

FAULTS: BADARG if ARG 2, when present, does not name an ES with an attached control-mark point

EXAMPLES: (..34,SAM) returning 101 as its value indicates that the status of the control mark point associated with the ES SAM is: <enable bit> = 1, <delivery> = 0 (<highest priority>), and <demand bit> = 1.

MACRO: SET CONTROL BUFFER STATUS - SETCBS

CLASS: Intra-process control primitive

ARG 1: ..35

ARG 2: <symbol>

ARG 3: [<integer>]

ARG 4: [<integer>]

ARG 5: <symbol> OPTIONAL

DESCRIPTION: The "arm" status of the nth (ARG 3) buffer of the control point in the control resource named by ARG 2, attached to the expression state named by ARG 5 is set or reset as ARG 4 is 1 or 0 respectively. If ARG 5 is absent, the active control point is designated. Note that this primitive will generate a type fault on ARG 2 if it is not generated by the <primary access declaration> of a control resource.

VALUE: ARG 2 itself

SIDE EFFECTS: A meta-message is issued to the system to create or destroy the corresponding system buffer if its status has changed.

FAULTS: BADARG if ARG 3 indexes a non-existent buffer, if ARG 4 is not 0 or 1, if ARG 5 does not name an expression state, if no control point is attached to the normed expression state, or if ARG 2 does not name a CONTROL type resource.

EXAMPLES: (..35, CONTROL, 5%, 1%, ESA) will arm the fifth buffer of the control point associated with ESA.
(..35, CONTROL, 2%, 0%) will disarm the second buffer of the currently active control point.

MACRO: READ CONTROL BUFFER STATUS - RDCBS

CLASS: Intra-process control primitive

ARG 1: ..36

ARG 2: <symbol>

ARG 3: [<integer>]

ARG 4: <symbol> OPTIONAL

DESCRIPTION: The "arm" status of the n th (ARG 3) buffer of the control point (in the control resource named by ARG 2) attached to the expression state named by ARG 4 is read and returned as [0] if disarmed, and [1] if armed. If ARG 4 is absent, the currently active control point is designated.

VALUE: The status of the designated buffer, [0] or [1].

SIDE EFFECTS: None

FAULTS: BADARG if ARG 3 indexes a non-existent buffer, if ARG 2 does not name a control resource, if ARG 4 does not name an expression state, or if no control point is attached to the named expression state.

SIDE EFFECTS: None

FAULTS: BADARG If ARG 3 indexes a non-existent buffer, if ARG 2 does not name a control resource, if ARG 4 does not name an expression state, or if no control point is attached to the named expression state.

MACRO: TRANSMIT MESSAGE - XMIT

CLASS: Transmission primitive

ARG 1: ..37

ARG 2: <symbolic>

ARG 3: <literal>

DESCRIPTION: ARG 2 must name a "RECEIVER type" resource <allocable object>. The <literal> named by ARG 3 will be sent as a message to the control point buffer 1 entified in the resource <allocable object> named by ARG 2. Note that this primitive will fault on ARG 2 reference if not generated from the <primary access declaration> of a RECEIVER resource.

VALUE: ARG 3 itself

SIDE EFFECTS: A message is transmitted as described above

FAULTS: BADARG if ARG 2 does not name the <accountable object> of a RECEIVER resource

MACRO:	READ MARK BUFFER - RDMBUF
CLASS:	Intra-process reception primitive
ARG 1:	..38
DESCRIPTION:	The <object> in the buffer of control mark point associated with the active expression state is returned as the value. The buffer is then marked "empty".
VALUE:	The <object> as described above
SIDE EFFECTS:	Removal of the contents of the mark point buffer
FAULTS:	NOOBJ if buffer is empty

MACRO:	DECLARE RESOURCE SET ENTRY - DECRES
CLASS:	Intra Process Primitive on Resources
ARG 1:	..40
ARG 2:	[<symbol>]
ARG 3:	[<integer>]
DESCRIPTION:	Creates a Resource Set entry, with name supplied by ARG 2, to accept resource objects. Installs number of associated promotion mechanism declaration name entries as specified by ARG 3. The resource status "presence" entry is flagged as empty.
VALUE:	ARG 2 Itself
FAULTS:	BADARG if symbol named by ARG 2 already exists in the Resource Set.
SIDE EFFECTS:	Environment structure modification.

MACRO:	DESTROY RESOURCE SET ENTRY - DSTRES
CLASS:	Intra Process Primitive on Resources
ARG 1:	..41
ARG 2:	[<symbol>]
DESCRIPTION:	Deletes the ARG 2 named entry and the associated protection declarations from the GDL.
VALUE:	ARG 2 Itself
FAULTS:	BADARG if Resource Set entry creator status is not Local. BADARG if reference count $\neq 0$. BADARG if Resource Set entry presence status is "non-empty".
SIDE EFFECTS:	Environment structure modification.

MACRO:	INSTALL RESOURCE - INSRES
CLASS:	Intra Process Primitive on Resources
ARG 1:	..42
ARG 2:	<accountable object declaration>
ARG 3:	<object>
ARG 4:	<object>
ARG n:	<object> (OPTIONAL for $n \geq 5$)
DESCRIPTION:	ARG 3 object becomes an accountable object associated with the ARG 2 declaration previously created in Resource Set. Installs ARG 4 through ARG n as protection declaration to resource object. The resource status is marked as "present" and "local".
VALUE:	ARG 2 Itself
FAULTS:	BADARG if entry Resource Set Presence bit is set. BADARG if number of protection declaration does not match number associated with the resource object in the declarative list.
SIDE EFFECTS:	Environment structure modification.

MACRO: DESTROY MARK - DESMRK

CLASS: Intra Process Primitive on Control Resource.

ARG 1: ..45

ARG 2: ⌊<symbol>⌋

DESCRIPTION: Destroys the rightmost mark associated with the resource referenced by ARG 2. If the rightmost mark is associated with an <allocable object> the mark is made null. If the mark is not associated with an <allocable object>

VALUE: ARG 2 Itself.

FAULTS: BADARG if no mark associated with ARG 2.

SIDE EFFECTS: Environment structure modification.

MACRO: ALLOCATE OBJECT - ALLOBJ

CLASS: Inter Process Primitive on Resource <object>.

ARG 1: ..46

ARG 2: ⌊<symbol>⌋

ARG 3: ⌊<symbol>⌋

ARG 4: <object> OPTIONAL

DESCRIPTION: Allocates named resource object, specified by ARG 2, to child process named by ARG 3. Protection declaration transmission in implicit if named child process is not listed in the audit trail as a current holder of an allocable object of this resource. Marks the Global Declaration List entry, in the child process, specified by ARG 2 as "GLOBALLY" installed resource. ARG 4 specifies an additional protection mechanism to be allocated along with resource object.

VALUE: ARG 2 Itself

FAULTS: BADARG if no allocable object available.
NAB if no acknowledgement butter available.
NOMOD if requested resource is currently in non-modifiable state.

SIDE EFFECTS: Places message containing resource object and associated protection declarations in Interface Buffer for transmission to child process.

MACRO: RETURN OBJECT - RETOBJ

CLASS: Inter Process Primitive on Resource <object>.

ARG 1: ..47

ARG 2: [<symbol>]

ARG 3: [<flag bit>]

DESCRIPTION: Returns named resource object, specified by ARG 2, to parent process. ARG 3 contains the flag which specified "One"-0 or "All"-1 resource object are to be returned. If ARG 3 specified "All", then the Global Declaration List entry, specified by ARG 2, will be marked as null.

VALUE: ARG 2 Itself

FAULTS: BADARG if no allocable object exists to return.
NAB if no acknowledgement buffer exists.
NOMOD if requested resource is currently in non-modifiable state.
BADARG if on "Return All", any object associated with named resource is currently allocated.

SIDE EFFECTS: Places message containing resource object to be returned in Interface Buffer for transmission to parent process.

MACRO: PRE-EMPT OBJECT - PREOBJ

CLASS: Inter Process Primitive on Resource <object>.

ARG 1: ..48

ARG 2: [<symbol>]

ARG 3: [<symbol>]

ARG 4: [<flag bit>]

DESCRIPTION: Pre-empts resource objects named by ARG 2 from child process named by ARG 3. As specified by ARG 4, message will contain flag bit specifying "One"-0 or "All"-1 resource objects to be pre-empted. If ARG 4 specified "All", then the Accountable Objects processor in the child will mark the Global Declaration List entry, specified by ARG 2, as well.

VALUE: ARG 2 Itself

FAULTS: BADARG if no resource object in audit trail of named child process. NAB if no acknowledgement buffer available. NOMOD if requested resource is currently in nonmodifiable state.

SIDE EFFECTS: Place message containing resource object name to be pre-empted from child process in Interface Buffer.

MACRO: CREATE PROCESS - CRPROC

CLASS: Intra Process Primitive on Process.

ARG 1: ..50

ARG 2: [<symbol>]

ARG 3: [<symbol>]

ARG 4: [<integer>]

ARG 5: [<symbol>]

ARG 6: [<integer>]

ARG i: [<symbol>] OPTIONAL

ARG i + 1: [<integer>] OPTIONAL

DESCRIPTION: Creates process named by ARG 2, formed by a basic Accountable Objects expression state, containing a dictionary segment called the Resource Set of the process. ARG 3 specifies the declaration entry and ARG 4 the number of protection decls for a control point resource. A mark is associated with a null ARG 3 resource object and distinguishes the Expression State named by ARG 5. The expression state name (ARG 5), and the number of associated protection declarations (ARG 6) are entered into the Global Declaration List of this new process. ARGi and ARG i+1 specify additional resource object declaration names and number of associated protection declarations as optional pairs. The child process name is entered as a relative entry in the current process space. In the child process relative entry the name of the PR buffer associated with that child is put there by PR at process birth.

VALUE: ARG 2 Itself

FAULTS: BADARG if duplicate process name exists.
BADARG if PR channel busy.
NAB if no acknowledgement buffer exists.

SIDE EFFECTS: Creates a message to be placed in the Interface Buffer, addressed to PR, to install the new process in this system. PR attaches the process in the system, by establishing the AO channel termination buffers. PR creates a channel termination buffer for the newly created process within PR.

MACRO: MOVE PROCESS - MVPROC

CLASS: Intra Process Primitive on Process

ARG 1: ..51

ARG 2: [<symbol>]

DESCRIPTION: Moves current process to system named by ARG 2.

VALUE: ARG 2 Itself

FAULTS: NAB if no acknowledgement buffer exists.
BADARG if no system named by ARG 2 exists.
NOMOD if process currently waiting on completion of an outstanding AO operation.
BADARG if process not prepared for movement, is, all required meta-opa not fulfilled.

SIDE EFFECTS: Creates a message to be placed in the Interface Buffer, addressed to PR, to move this process to the system named by ARG 2. Issues meta-op to this system to disconnect and excise this process from this system.

MACRO: DESTROY PROCESS - DSPROC

CLASS: Inter-Process Primitive on Process.

ARG 1: ..52

ARG 2: [<symbol>]

DESCRIPTION: Removes process referenced by ARG 2 from the process tree, at the request of parent process. Removes child name from the relatives declaration of the parent process GDL. A message is issued to the child AO to return all resources, AO checks it out and sends process to PR.

VALUE: ARG 2 Itself

FAULTS: BADARG if ARG 2 is not a child process.
NAB if no acknowledgement buffer exists.

SIDE EFFECTS: Process tree modification.

BIBLIOGRAPHY

C Cowan, George, "A General Structure for Resource Management in a Computer Network", Ph.D. thesis, University of Wisconsin, expected December 1974.

K Kramer, J. F., "A General Structure for Uncooperative Processes Distributed over a System Network", Ph.D. Thesis, University of Wisconsin, June 1973.

KF Kramer, J. F. and D. R. Fitzwater", The Architecture of a Machine Independent Network Operating System for Hierarchial Delegation of Authority", to be published.

M Mooers, C. N., "TRAC, A Procedure - Describing Language for the Reactive Typewriter", CACM, Vol. 9 (March 1966).

N Naur, P., et. al., "Report on the Algorithmic Language ALGOL 60", CACM, Vol. 3 (May 1960).