CMU-CS-79-137

The Symbolic Manipulation of Computer Descriptions

The ISPS Computer Description Language

Mario R. Barbacci
Gary E. Barnes
Roderic G. Cattell
Daniel P. Siewiorek

Departments of Computer Science
and Electrical Engineering
Carnegie-Mellon University
14 August 1977
6 March 1978
16 August 1979

The development of ISPS is part of the research on the Symbolic Manipulation of Computer Descriptions sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Table of Contents

ISPS Reference Manual

17. References

Appendix I. Syntax Charts

Index

List of Figures

List of Tables

The Symbolic Manipulation of Computer Descriptions

Designers make use of notations and languages as abstraction building tools. The meaning of these abstractions is based on a set of predefined notions on the domain of problems that the designers attempt to solve. In the Symbolic Manipulation of Computer Descriptions (SMCD) project we are attempting to design and build systems that operate relative to computer descriptions. Thus we need abstractions to describe computers.

When one tries to design a problem oriented language, one can use different notations for each problem area and as new problem areas become the focus of our research we simply develop new languages as needed. Alternatively, one can make a guess and develop a language that incorporates every possible abstraction that anybody might ever need. It is easy to see why neither of these solutions is satisfactory. If we use different languages we need to translate machine descriptions developed in other notations into our own. Verifying the translation is akin to testing the equivalence of algorithms, an unsolvable problem in general. Any certification that might have been painstakingly obtained is now lost if we can not verify the equivalence of the descriptions.

The second approach tends to yield complicated, hard to understand, and sometimes unimplementable languages. All users must pay the price of the inefficiencies introduced to cope with a huge set of abstractions, even if the application one has in mind requires a limited subset of what the language offers.

ISPS is a kernel language which provides the users with the tools to define application dependent abstractions around a core notation. It is the applications that define the meaning of the abstractions. This alleviates the problem of accommodating new, unforeseen requirements by providing users with a set of tools: a language and a parser, and a mechanism by which the meaning of the descriptions can be specified, modified, or extended. We have avoided the problem of the inefficiency of an umbrella language by partitioning the implementation of application dependent semantics among the users. Only those abstractions needed for an active area of research are implemented and the body of expertise centered around the manipulation of ISPS descriptions can grow gradually.

Mario R. Barbacci, 14 August 1977

Preface to the Second Edition

A number of typographical errors have been corrected in this second edition of the manual. A few comments have been added to help clarify the meaning of some features. The changes however, are of a minor nature and users of the first edition of the manual (8/14/77) do not

need a new copy. The ISPS readers that provided me with valuable feedback are too numerous to mention. I am grateful for their comments.

MRB, 6 march 1978

Preface to the Third Edition

This is the third edition of the ISPS reference manual. It describes the features implemented in version 5 of the ISPS parser. The main differences with the previous release are: a) the elimination of one construct (concatenated mappings) and, b) and the introduction of several new operators and predeclared entities. In addition to presenting the new language features, many sections have been rewritten to clarify the language constructs and their intended use.

The manual now includes a set of charts describing the syntax in a pictorial manner. It is hoped that these will complement the BNF in presenting the syntax of the language.

The previous edition of the manual mentioned an "applications manual" and many users requested copies. Unfortunately, the "applications manual" was never intended to be a single, monolithic document, but rather an expanding set of documents describing systems and programs making use of ISPS. Some of these documents are included in the software distribution tape. Others are to be obtained directly from the authors or maintainers of the applications programs.

The previous edition of the manual included an introductory chapter. This chapter has been eliminated since it is available as a separate technical report [Barbacci, 1978] and as an appendix in a published book [Bell, 1978].

MRB, 16 August 1979

1. Introduction

The ISP notation was first introduced by Bell&Newell [Bell,1971] as a formalism to describe the programming level in the hierarchy of digital systems descriptions. At the programming level a computer is described in terms of data types, data operations, and an interpretation rule. The interpreter is an algorithm that defines the sequences of operations performed by the machine. These operations are encoded in a particular data type: the instruction, and they operate on other data types encoded in the memory and registers of the machine. The data types are stored or transmitted in data carriers (memories, registers, and data paths). These values are transformed or operated upon by the data operators (functional units), controlled by a network of clocks and sequential circuits. All of these components are defined in terms of a lower level of computer descriptions, the Register Transfer level.

Although ISP is oriented towards the description of Instruction Set Processors, it contains a fair number of constructs which can be used to describe a large class of register transfer systems (digital computers are a subset of the latter, namely, those systems that fetch, decode and execute instructions).

The design philosophy of ISPS was guided by two principles, flexibility and simplicity. Specifically, it was desired to design a computer description language that would be appropriate for diverse applications: automated design, simulation (for both software development and hardware debugging), and automatic generation of machine relative software (in particular, compiler-compilers). Thus, although ISPS can be viewed as a programming language, the aim of the notation is to describe computers and other digital systems, not necessarily general computational algorithms.

The ISPS language is parsed by a 'compiler' which runs on a PDP-10. This is not a compiler in the normal sense; its output is a parse tree, which is used as input by the various aforementioned application programs.

The definition of what constitutes a 'correct' ISPS description depends to some extent on the nature of the application programs using the machine description. An assembler generator might, for instance, require the specification of the instruction mnemonics but it might not have any use for the specification of the memory technology. The situation is reversed when a design automation system uses the same parse tree. A compiler-compiler system might be interested in the 'cost' of each instruction in order to generate optimal code. For details on these and other applications, see [Barbacci, 1979].

To allow the coexistence of multiple applications, ISPS provides an extension facility for the specification of application dependent information. This information is attached to the parse trees and can be easily retrieved by the application programs. Because of the open

ended nature of the application dependent information, the parser can only perform syntactic analysis of the extensions. Relatively little can be done at parse time with regard to the semantic analysis and the bulk of the semantic analysis of the extensions thus lies in the domain of the application areas[1].

We are indebted to many individuals, at CMU and elsewhere, for their comments, criticisms, and encouragement. The RT-CAD group at CMU and the meetings of the AMD Working Group provided invaluable feedback to the designers of the language. The following individuals deserve special thanks: Steve Crocker (USC-ISI), Lloyd Dickman (DEC), Vittal Kini (CMU), Barry Press (TRW), Don Thomas (CMU), and Andries Van Dam (Brown University).

[1] As experience with the language grows, the semantic knowledge built into the parser will be augmented to incorporate those aspects that are common to all applications or which can result in contradictory assumptions by the users of the machine description.

2. Syntactic Conventions

The syntax of the language is defined in the Backus-Normal-Form (BNF) meta-notation. The characters '::=' separate the name of a production from the sequence of terminals and non-terminals which define the production. Alternatives sequences are separated by '|'.

All production names are written in bold face (e.g. **c-expression**). Keywords and reserved identifiers are written in upper case (e.g. BEGIN). Bear in mind however, that ISPS makes no distinction between upper and lower case letters, thus in an actual description 'BEGIN' and 'begin', and even 'BeGiN' are all equivalent.

In order to keep the number of BNF productions down to a level which does not impair the readability of this manual, the following meta-convention will be used: A production name of the form 'x-LIST^{y}' stands for a sequence of, at least one, instances of x, separated by 'y', where 'y' can be any character, including NULL (this allows for the specification of lists without any special delimiter). i.e.:

Z ::= X-LIST, is equivalent to: Z ::= X | Z ; X

By the same token, a production of the form 'x-LIST^{y}-LIST^{z}' stands for a sequence of, at least one, instances of 'x-LIST^{y}', separated by 'z'.

3. Character Set, Identifiers and Constants

3.1. ISPS Character Set

The character set used in ISPS is essentially the full 7-bit ASCII character set. Upper and lower case letters are considered to be equivalent (the ISPS parser maps all letters to their upper case form.) Most other characters are taken literally with no mapping performed on them. Table 3-1 depicts the non-alphanumeric characters and their meaning in ISPS.

Octal	Char.	Use	
041	!	Indicates a comment	(See [3.4])
042	"	Indicates a hexadecimal constant	(See [3.3])
043	#	Indicates an octal constant	(See [3.3])
047	'	Indicates a binary constant	(See [3.3])
050	(	Used in blocks and expressions	(See [5.2, 7])
051	)	Used in blocks and expressions	(See [5.2, 7])
052	*	Multiplication operator	(See [6.2.5])
053	+	Addition operator	(See [6.2.1, 6.2.6])
054	,	Used as a list separator.	(See [4.2, 4.3, 5.4, 7, 9])
055	-	Subtraction operator	(See [6.2.1, 6.2.6])
056	.	Used in identifiers	(See [3.2])
057	/	Division operator	(See [6.2.5])
072	:	Used to indicate a range of values	(See [3.7])
073	;	Concurrent Action and Qualifier separator	(See [5, 9])
074	<	Used to describe a bit structure	(See [4.2, 7])
075	=	Logical transfer operator	(See [6.3])
076	>	Used to describe a bit structure	(See [4.2, 7])
077	?	Used in Constants	(See [3.3.3, 5.4.2])
100	@	Concatenation operator	(See [6.2.3])
133	[	Used to describe a word structure	(See [4.2, 7])
134	\	Used in Aliases	(See [3.5])
135	]	Used to describe a word structure	(See [4.2, 7])
137	_	Logical transfer operator	(See [6.3])
173	{	Used in Qualifiers	(See [9])
174	|	Used to quote strings	(See [3.6, 5.2, 9, 10])
175	}	Used in Qualifiers	(See [9])

Table 3-1: Non-alphanumeric Characters

Tabs, form feeds, blank lines, etc. may be used anywhere a space may be used.

In addition to the characters shown in Table 3-1, ISPS makes use of certain special

characters outside the ASCII character set. These special characters have obvious transliterations in terms of multiple ASCII characters, as shown in Table 3-2

Char.	Use	
:=	Used in declarations and labels	(See [4.1, 5.3, 5.4, 10])
=>	Used in conditional actions	(See [5.4])
<=	Arithmetic transfer operator	(See [6.3])
**	Section Head Delimiter	(See [4.3])

Table 3-2: Special Characters in ISPS

3.2. Identifiers

Identifiers in ISPS are made up of the following characters: A-Z, a-z, 0-9, and '.' Upper and lower case letters are equivalent.

Identifiers must start with a letter and may be of any length (the current implementation limits **identifiers** to be up to 80 characters long).

Examples:

```
NAME
NaMe
This.is.a.long.identifier
c1d3e5
```

3.3. Constants

A **constant** is a sequence of characters in some alphabet determined by the base of the **constant**. The default base is ten; any **constant** which appears without a base indicator is considered to be decimal. Base eight **constants** are preceded by *. Base two **constants** are preceded by '. Base sixteen **constants** are preceded by ".

3.3.1. Constant Alphabets

The alphabets for the predefined bases in ISPS are depicted in Table 3-3.

Base	Prefix	Alphabet
2	'	0,1,?
8	#	0,1,2,3,4,5,6,7,?
10	none	0,1,2,3,4,5,6,7,8,9
16	"	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,?

Table 3-3: Representation of Constants

3.3.2. Kilo and Mega Multipliers

For convenience, any constant may be appended with a sequence of Ks, which acts as a multiplier of value = 1024. Thus 1K is equivalent to 1024, 4K is equivalent to 4096, 1KK is equivalent to 1024K or 1048576. Similarly, a sequence of Ms can be used as a multiplier of value = 1048576.

3.3.3. Don't Care Digits

The character '?' can be used in a binary, octal, or hexadecimal constant to specify a don't care digit. Its presence stands for any digit in the corresponding alphabet. Don't care digits are a notational convenience for writing partially specified 'bit patterns'. In the current implementation the use of '?' is restricted to constants used to label DECODE alternatives (See [5.4.2]).

Examples:

```
"1000    ! base 16
4096     ! base 10
#10000   ! base 8
'1000    ! base 2
```

3.3.4. Length of Constants

Constants have two properties. The first of these is the base they are in, the second is their length. The length of a constant is measured in bits, according to the rules shown in Table 3-4.

Base	Length of Constants
10	Decimal constants are one bit longer than the smallest number of bits needed to represent its value.
2	Binary constants have one bit for each digit explicitly written.
8	Octal constants have 3 bits for each digit explicitly written.
16	Hexadecimal constants have 4 bits for each digit explicitly written.

Table 3-4: Length of Constants

Examples:

```
"1000    ! 16 bits
4095     ! 13 bits
#17      ! 6 bits
0        ! 2 bits
"A1      ! 8 bits
"00F     ! 12 bits
#10000   ! 15 bits
'101     ! 3 bits
'00??1   ! 5 bits
#?2      ! 6 bits
```

ISPS provides means to specify constant bit patterns whose length do not follow the above length rules. These are described under c-terms (See [7]).

3.4. Comments

A comment is indicated by a '!'. Everything from the '!' to the end of the line in which it appears is treated as a commentary and is not parsed. Comments appear in the parse tree (See [14.4]) and can be used by the application programs:

Example: `! This is a comment`

3.5. Alias

It is possible to attach an **alias** to any **identifier**. The **alias** is usually a long form of the **identifier** which is too long to use conveniently, e.g.:

Examples:

```
PC\Program.Counter
ALNFAR\A.Long.Name.For.A.Register
```

It is also possible to attach an **alias** to any **constant**. This **alias** is a convenient way of attaching a special meaning to the **constant**, e.g.:

Examples:

```
'0110\Priority.Mask
#204\AND.Instruction
```

Any number of **aliases** can be specified, separated by '\'. Syntactically, an **alias** is an **identifier**. Semantically, an **alias** is treated as commentary information by the parser. i.e. PROGRAM.COUNTER is not interchangeable with PC.

Aliases are kept in the parse tree (See [14.4]) for the benefit of the application programs.

3.6. Quoted Text

There are several places in an ISPS description where an arbitrary ASCII string is valid (e.g.: qualifiers, see [9]). These strings are mainly for the benefit of the application programs that operate on the ISPS parse trees. A **quoted-text** is a string of characters enclosed between '|' and '|'. A '|' inside the **quoted-text** is represented by '||':

Example: `|This a a random string with a || in it|`

3.7. Name Pairs

In ISPS there are several uses for a list of **constants**. A **name-pair** is an abbreviated notation for a list of consecutive **constants**.

```
name-pair ::=   constant |                                   (See [3.3])
                constant : constant
```

Examples: 3:5 stands for 3,4,5; 7:4 stands for 7,6,5,4

Name-pairs are used, among other things, to name the bits of a carrier. A **name-pair-LIST** (a list of **name-pairs** separated by ',') is used to label the elements of a list of DECODE alternatives (See [5.4]).

Examples:

```
3,5:7,4                 (equivalent to 3,5,6,7,4, in that order)
15:13,12,11:10          (equivalent to 15:10 or 15,14,13,12,11,10)
```

4. ISPS Descriptions

ISPS describes the interface (i.e. external structure) and the behavior of abstract hardware units (called entities in the language). The interface describes the number and types of carriers used to store and transmit information between the units. The behavioral aspects of the unit are described by procedures which specify the sequence of control and data operations in the machine.

Formally, in ISPS we define a digital system as a network of entities. These entities can have different structural and behavioral properties. Thus, the declaration of the main memory of a computer must specify the number of words and the number of bits per word. An arithmetic unit must specify the different operations it can perform. Other components share both types of properties. For instance one could describe a bus as an entity with a structure (e.g. the bus lines) and a function (e.g. the arbitration mechanism).

The exact physical implementation, the data storage and transmission mechanisms, the mechanisms used to activate and terminate the execution of operations, the synchronization and timing characteristics, etc. can be specified via qualifiers (See [9]). This type of information is not required by the parser. That is, an ISPS description does not need to specify the actual implementation or even the organization of the data paths.

4.1. Declarations

```
ISPS-declaration ::=   e-declaration
e-declaration ::=      e-head |                          (See [4.2])
                       e-head := e-body |                (See [4.3])
                       other-declarations                (See [10])
```

An **ISPS-declaration** is the minimal parsing unit. For the sake of explanation we can call it a 'program'. An entity declaration (**e-declaration**) defines a hardware component which might have a structure and exhibit some behavior. The entity head (**e-head**, see [4.2]) defines the structural properties. The entity body (**e-body**, see [4.3]), if present, defines the behavioral properties.

4.2. Structure

```
e-head ::=          identifier fc-set fs-set              (See [3.2])
fc-set ::=          nil | ( ) | ( e-head-LIST+ )
fs-set ::=          nil | bit-fs-set | word-fs-set bit-fs-set
word-fs-set ::=     [ name-pair ]                         (See [3.7])
bit-fs-set ::=      < > | < name-pair >
```

An **e-head** defines the structural aspects of a declaration. It consists of several parts,

some of which may be optional:

Component	Meaning
identifier	The **identifier** distinguishes the entity from other entities defined at the same level or scope (See [4.4]). It must always be present.
fc-set	The formal connection set (**fc-set**) defines an interface for connecting entities. Syntactically, it is a (possibly empty) list of carriers (**e-heads**) playing the role of 'formal parameters'. The **fc-set** is optional.
fs-set	The formal structure set (**fs-set**) defines a carrier and it may consist of a single 'word' or an array of 'words' (a memory). The **fs-set** is optional.
word-fs-set	The word formal structure set (**word-fs-set**) defines the structure of an array of 'words'. The names of the words are specified inside '[' and ']'. The **word-fs-set** is optional.
bit-fs-set	The bit formal structure set (bit-fs-set) defines the structure of a single word. The names of the bits are specified inside '<' and '>'. The **bit-fs-set** must be specified if the **fs-set** is present.
name-pair	The elements of the **name-pairs** (the dimensions) specify a naming convention for the 'words' and 'bits' of a carrier. The **name-pair** must be present in a **word-fs-set**, it is optional in a **bit-fs-set** (an empty **bit-fs-set** (<>) stands for a single, unnamed bit.)

4.3. Behavior

e-body : : =	BEGIN **section**-LIST END | (**section**-LIST) |	
	BEGIN **b-expression** END | (**b-expression**) |	(See [5])
	e-head	(See [4.2])
section : : =	**section-header e-declaration**-LIST,	(See [4.1])
section-header : : =	☆☆ **identifier** ☆☆	(See [3.2])

An entity body (**e-body**) defines the behavior of an entity. There are three kinds of 'bodies':

Body Type	Explanation
section-LIST	The most general case of an **e-body** consists of a list of **sections**, each consisting of a **section-header** followed by a list of **e-declarations**, local to the body. The declarations inside the **sections** can be of arbitrary complexity. They can in fact, have bodies with local **sections** to any level of nesting.

Declarations are grouped in **sections** as an abstraction

mechanism. Application programs which manipulate ISPS parse trees might require specific **sections** to be present while possibly ignoring others.

b-expression Simpler bodies are defined by a **b-expression** (a behavioral expression, see [5]) which can be thought of as a sequential or combinational network, depending on the nature of the operations used and the implementation thereof.

e-head The third type of **e-body** is used to define or map alternative structures and naming conventions over previously declared carriers. The bit (or word) names used on the left hand side of a structure mapping are independent from the bit or word names used on the right hand side. Both sides of a mapping must, however, specify structures of the same size (# words * # bits/word). The equivalence between the bits of the right hand side and the bits in the left hand side is obtained by aligning the leftmost bit of the leftmost word of the left hand side with the leftmost bit of the leftmost word of the right hand side.

4.4. Scope of Declarations

ISPS declarations follow the same scope rules used in Algol and other programming languages. Declarations must be unique within a **section**-LIST defining the body of an entity. **Sections** do not define scopes, i.e. declarations in separate **sections** of the same **section**-LIST must have unique **identifiers**.

Entities declared inside another entity are not accessible outside the enclosing entity. Internal entities are 'own' (in the Algol sense), thus carriers used to describe storage elements preserve their values across activations of the enclosing entity.

Labels used to name **labelled-actions** (See [5.3]) are not considered to be declarations in the normal sense. The label (an **identifier**) is known and available only inside the **labelled-action** and no conflicts arise from the use of the same label for several concurrent or sequential **labelled-actions**.

4.5. Examples

```
Ir\Instruction.Register<0:31>
```

This declaration defines a 'register' (IR) whose structure consists of 32 bits (0,1, ...,30,31). The elements of the **name-pair** 0:31 specify the names of the bits.

```
Mp[0:255]<7:0>
```

This declaration defines a 'memory' (M) whose structure consists of 256 words, each 8 bits long. The words are named 0,1,2,..255 while the bits inside each word are named 7,6,...,1,0.

```
VMA\Virtual.Memory.Address<13:35>
```

This declaration shows that the bit (and word) names can be specified in ascending or descending order. In fact, the name-pairs do not even have to begin or end on 0. The VMA carrier is declared to be 23 bits long, the bits named as 13,14,....,34,35 (the example was derived from the DEC PDP-10 Virtual Memory Address format; the VMA carrier is loaded with bits 13 through 35 of the Instruction Register, thus the bit names.)

```
Alu(Areg<0:15>,Breg<0:15>)<0:16>
```

The example defines the structure of a 'functional unit' (ALU) which consists of two interface carriers (AREG<0:15> and BREG<0:15>) and one 'result' carrier, (ALU<0:16>). The word 'result' is being used in an informal sense. ALU<0:16> is just another carrier and be can read or written by other entities as well as by ALU itself.

```
MARK1 :=                                    ! Manchester University Mark-1
        Begin
        ** Memory.State **
        M[0:8191]<31:0>,

        ** Processor.State **
        PI\Present.Instruction<15:0>,
        CR\Control.Register<12:0>,
        Acc\Accumulator<31:0>,

        ** Instruction.Execution **
        I.Cycle :=
                Begin
                . . . . . .
                End
        End
```

The example depicts the body of the declaration of an entity, in this case a minicomputer. The header of the declaration (MARK1) does not specify an structure. The body or behavioral part consists of a list of declarations for the memory, registers, and operations of the machine. These declarations are (arbitrarily) grouped into three sections (** MEMORY.STATE **, ** PROCESSOR.STATE **, and ** INSTRUCTION.EXECUTION **). The last

section consists of a single entity (I.CYCLE) which specifies the sequence of data and control operations.

```
I\instruction<15:0>,                          ! PDP-11 Instruction Format
        bop\binary.operation<2:0>       := I<14:12>,
        s\source.field<5:0>             := I<11:6>,
                sm\source.mode<1:0>             := s<5:4>,
                sd\source.deferred<>            := s<3>,
                srcreg\source.reg<2:0>          := s<2:0>,
        d\destination.field<5:0>        := I<5:0>,
                dm\destination.mode<1:0>        := d<5:4>,
                dd\destination.deferred<>       := d<3>,
                desreg\destination.reg<2:0>     := d<2:0>,
```

In the above example, several fields of I (The PDP-11 Instruction Register) have been defined as if they were independent 'registers' (i.e. each field has its own name, with an optional alias, and a structure or dimension specification).

```
    CCodes[0:3]<>  := PSW<15:18>,                ! S360 Condition Codes
```

The last example shows how different structures can be mapped on top of a previously declared carrier. CCODES is defined as an array of 4 1-bit carriers. Thus, one can access the bits in the field PSW<15:18> using two alternative structures (i.e. an array of 1-bit carriers or a 4-bit field). The equivalence of bits is as follows: The leftmost bit of word 0 of CCODES corresponds to bit 15 of PSW. Since this is the only bit of word 0, we continue on word 1, whose bit corresponds to bit 16 of PSW, etc. etc.

5. Behavioral Expressions

b-expression : : =	**s-action**	
s-action : : =	**p-action-LIST**^{NEXT}	
p-action : : =	**action-LIST**;	(See [5.1])

A behavioral expression (**b-expression**) defines the behavior of an entity. **b-expressions** are built by specifying the sequence of transformations and transfers of values stored in carriers. Simple **b-expressions** (**actions**) can be combined to build larger **b-expressions** by activating them in sequence (**s-actions** separated by NEXTs) or concurrently (**p-actions** separated by ';').

No synchronization must be assumed between parallel **actions**. **Actions** separated by ';' are considered to be order independent and can be executed in any fashion. The only requirement is that parallel **actions** are completed before proceeding beyond the following NEXT separator.

Order independence refers to the order of evaluation of the actions, it does not refer to the order of initiation of the actions. Parallel actions, separated by ';' are initiated concurrently, their execution can proceed in any order.

Notice that this does not imply that the same results should be obtained from all implementations. This is particularly true when the concurrent actions contain control operators that modify the flow of control. It is not necessarily the case that the writer of an ISPS description is aware of the consequences of specifying concurrent operations separated by ';'. In general, it is an unsolvable problem to determine when two concurrent operations are meaningful or desirable. The ISPS Parser does not even try to check concurrent operations *Caveat Emptor*.

Example	Description
A=1; B=2 Next C=3	In this example the first two transfers are initiated in parallel and then, after their completion, the third one is performed.
...Next A=1; B=2 Next (C=3 Next D=4); E=5 Next...	A=1 and B=2 are performed in parallel. Then, the sequence C=3 followed by D=4 is performed in parallel with E=5.

5.1. Actions

action : : =	**block-action** |	(See [5.2])
	labelled-action |	(See [5.3])
	conditional-action |	(See [5.4])
	control-action |	(See [5.5])
	c-expression	(See [6])
block-action : : =	BEGIN **b-expression** END | (**b-expression**)	(See [5])
labelled-action : : =	**identifier : = action**	(See [3.2])

Actions are used to build complex behavioral expressions ranging from a primitive **c-expression**, to conditional or unconditional control flow operations, to a complex **b-expression** inside BEGIN/END or parentheses. The latter type can be used to build arbitrarily nested **b-expressions**.

5.2. Block Actions

A **block-action** consists of a list of sequential or concurrent **actions** (a **b-expression**) enclosed inside BEGIN/END or parentheses. The brackets are used to specify an order of execution different from that implied by the precedence of the sequencing (';' and 'NEXT') and data operations.

The brackets can be optionally followed by a **quoted-text** or block name to provide the reader with some degree of visual identification of the levels of nesting:

```
X :=
        Begin   | this is the outer block |
                . . . . .
                Begin   | this is the inner block |
                        . . . . .
                End     | this is the inner block |
                . . . . .
        End     | this is the outer block |
```

The **quoted-texts** attached to matching BEGIN/END or parentheses pairs must be identical. The parser will warn the user if that is not the case.

5.3. Labelled Actions

Actions may be labelled to allow the description of complex activities, including selection and premature termination or reinitialization of **actions**.

```
x := Begin . . . . . End
```

5.4. Conditional Actions

```
conditional-action ::=   IF c-expression => action |                    (See [6, 5.1])
                         DECODE c-expression => BEGIN numbered-action-LIST, END
                         DECODE c-expression => ( numbered-action-LIST, )
numbered-action ::=      action |
                         constant := action |                           (See [3.3])
                         name-pair := action |                          (See [3.7])
                         [ name-pair-LIST, ]  := action |
                         OTHERWISE := action
```

Two operators, IF and DECODE, are used to specify the selection of alternative **actions**, depending on the value stored in a carrier or computed from a **c-expression**.

If the value of the **c-expression** associated with an IF operation is non-zero ('true') the **action** following the => operator is executed, otherwise it is skipped. The **c-expression** is interpreted as an unsigned value (See [6.1]).

The **c-expression** associated with a DECODE operator is evaluated and its value used to select one of the **actions** specified in the **numbered-action-LIST,** following the => operator. The **c-expression** is interpreted as an unsigned value (See [6.1]).

5.4.1. DECODE Action Selectors

When the DECODE operation specifies a large number of alternatives, it is sometimes difficult for a reader to associate the alternatives with the values of the **c-expression** which selects them. In ISPS one can explicitly write the value of the **c-expression** associated with the action as a label-like action selector:

selector := **action**

Selector	Meaning
nil	If no selector is specified, the **actions** are assumed to have implicit selectors, given by the position of the **action** in the list of alternatives. The positions are numbered as 0,1,2,...
constant	A **constant** used to select an **action** identifies the value of the **c-expression** associated with the **action**.
name-pair	A **name-pair** used to select an **action** identifies a range of values of the **c-expression** associated with the **action**.
[**name-pair-LIST,**]	A list of **constants** and **name-pairs** can be enclosed inside '[' and ']' to indicate a non-consecutive list of values of the **c-expression** associated with the **action**.

OTHERWISE — The keyword OTHERWISE is used to specify a default **action** if the outcome of the **c-expression** is not covered by the other action-selectors.

If a value of the **c-expression** is covered by more than one action selector (either directly, as a **constant** or indirectly, as part of a **name-pair**) only the first **action** associated with the value is executed (i.e. exactly one **action** can be executed as a result of a DECODE operation).

It is a bad practice to mix **actions** with implicit and explicit action-selectors. The syntax allows it to handle the situation in which a designer is not yet sure of the proper **constant** action-selectors to use and wants to go ahead developing the ISPS description.

ALL outcomes of the **c-expression** must be accounted for. OTHERWISE must be used in some **action** if the number of **actions** is smaller than the number of possible values of the **c-expression.**

5.4.2. Don't Care Digits

The use of don't care digits ('?') in a **name-pair** of the form **constant:constant** used to label a DECODE alternative can be ambiguous. For instance, the range '*?5:*0?' could be construed as any of the following cases:

```
#05,#06,#07
#05,#04,#03,#02,#01,#00
#75,#74,#73,...,#11,#10,#07
#75,#74,#73,...,#02,#01,#00
```

More interesting patterns could be inferred from a more complicated example[2]:

```
'0?11?0 : '?11?0?
```

To avoid any ambiguity, the following (arbitrary) meaning has been attached to the use of don't care digits in a range:

1. The type of range (i.e. whether it is ascending or descending) is determined by assuming that all don't care digits have value '0, *0, or "0, as the case may be[3].

2. If the range is descending (i.e. the left **constant**, interpreted as in (1), is greater than the right **constant**, interpreted as in (1)), the extreme values or boundaries

---

[2] the determination of all possible ranges is left as an exercise to the reader!

[3] Remember that '?' is not allowed in a decimal constant.

of the range are defined by replacing all don't care digits in the left **constant** by '1, #7, "F, as the case may be, and replacing all don't care digits in the right **constant** by '0, #0, or "0, as the case may be.

3. If the range is ascending (i.e. the left **constant**, interpreted as in (1) is lesser that the right **constant**, interpreted as in (1)), the extreme values or boundaries of the range are defined by replacing all don't care digits in the left **constant** by '0, #0, "0, as the case may be, and replacing all don't care digits in the right **constant** by '1, #7, or "F, as the case may be.

4. The range consists of all consecutive **constants** contained between the boundaries determined above, including the boundaries. Notice that steps (2) and (3) are designed to define the largest possible range within the limitations imposed by step (1).

5.4.3. Examples

```
IF Acc EQL X => PC=PC+2
```

In this example, the operator EQL (See [6.2.7]) defines a 1-bit result ('0 stands for false, '1 for true). Depending on the value of this bit, PC is incremented (1) or not (0).

```
If Z<4:7> => Begin . . . . . End
```

This example shows that in general, the **c-expression** does not have to be 1 bit long. The **action** following the '=>' will be executed if ANY bit in the Z carrier is 1 (i.e. Z ≠ 0).

```
Decode OP<1:0> =>
        Begin
        ACC=0,                          ! if OP<1:0> is 0
        ACC=ACC+M[Z],                   ! if OP<1:0> is 1
        M[Z]=ACC,                       ! if OP<1:0> is 2
        PC=M[Z]                         ! if OP<1:0> is 3
        End
```

One of the four actions listed inside the BEGIN/END pair is executed, depending on the value of OP (0, 1, 2, or 3).

```
Decode OP<1:0> =>
        Begin
        0 :=    ACC=0,                  ! if OP<1:0> is 0
        2 :=    M[Z]=ACC,               ! if OP<1:0> is 2
        1 :=    ACC=ACC+M[Z],           ! if OP<1:0> is 1
        3 :=    PC=M[Z]                 ! if OP<1:0> is 3
        End
```

Notice that in the example we have altered the order of the actions. If explicit action selectors are used, as in the example, one is free to write the actions in any order. For instance, when describing the instruction decoding in a computer, one might wish to group all the ADD instructions (half word, full word, double word, floating point, etc), followed by all the SUBTRACT instructions, etc. even though the operation codes are not consecutive.

```
Decode F =>
        Begin
        0 :=    CR = M[S],
        1 :=    CR = CR + M[S],
        2 :=    Acc = - M[S],
        3 :=    M[S] = Acc,
        4:5 :=  Acc = Acc - M[S],
        6 :=    If Acc Lss 0 => CR = CR + 1,
        7 :=    Stop(),
        End next
```

The above example is taken from the Manchester University MARK-I computer [Lavington,1975]. Notice that there are two operation codes (4 and 5) associated with the Subtract operation.

```
Decode Address =>                                   ! PDP-11
        Begin
        #17???? := Begin .... End,                  ! I/O Page
        #00???? := Begin .... End                   ! memory
        End
```

In the PDP-11, addresses in the range #170000:#177777 constitute the I/O page and are handled differently from those in the range #00000:#167777. The use of don't care digits simplifies the writing of the alternative selectors.

```
eadd\effective.address<0:11> :=                    ! PDP8 Effective Address
Begin
Decode pb =>                                              ! Page Zero Bit
        Begin
        0 := eadd ='00000 @ pa,                             ! Page Zero
        1 := eadd = last.pc<0:4> @ pa                     ! Other Pages
        End Next
If ib =>                                                  ! Indirect Bit
        Begin
        If eadd<0:8> Eql #001 => M[eadd] = M[eadd] + 1 Next      !Autoindex
        eadd = M[eadd]                                    ! Memory Fetch
        End
End,
```

Although we have not yet defined the data operations, it should not be difficult to understand the example. Notice the use of the carrier associated with EADD (EADD<0:11>) in the computation of the effective address. Algol-like scope rules are used in ISPS and non-local carriers can be accessed from inside a body (e.g. IB, M, PA etc.)

5.5. Control Actions

control-action ::= REPEAT **action** | (See [5.1])
LEAVE **identifier** | (See [3.2])
RESTART **identifier** |
RESUME **identifier** |
TERMINATE **identifier**

5.5.1. Loops

An **action** that must be executed repeatedly (a loop) can be described by the use of the REPEAT operator preceding the **action**:

Example:

```
ICycle :=                                        ! PDP-10 Instruction Cycle
        Begin
        REPEAT
                Begin
                IR = Memory[Pc] Next                          ! Instruction Fetch
                Pc = Pc + 1; VMA = IR<13:35> Next                 ! Increment PC
                EA = VMA()<18:35> Next       ! Effective Address Computation
                IExecute()                                ! Instruction Execution
                End
        End
```

5.5.2. Action Terminators

The LEAVE, TERMINATE, RESTART, and RESUME operators are used to terminate the execution of an **action**. The **action** is specified through its label (in the case of a **labelled-action**) or through the **identifier** used in the **e-head** (in case the **action** is the body of an entity).

There are several restrictions which govern the use of these operators:

1. The label or entity name associated with the operation is statically bound. That is, the **action** to which the operation refers is determined by the lexical nesting of the declarations.

2. If the **identifier** is an **action** label, the operation must be lexically nested in the body of the **labelled-action**. In other words, if a **labelled-action** invokes an activity, the body of the activity can not use the label in a control operation. This is a consequence of the scope rules (See [4.4]).

3. If the **identifier** is an entity name, in the case of the LEAVE, RESTART, and RESUME operations, the operation must be dynamically or statically nested inside the entity. In the case of the TERMINATE operation, the operation need not be nested inside the action to be terminated.

Operation	Meaning
LEAVE	The LEAVE operator is used to force the termination of **labelled-actions** and **e-bodies**. Any **actions** (other than PROCESSes, see [8.3]) initiated during the execution of the **action** to be terminated and not yet completed are also terminated by the LEAVE operator. PROCESSes activated by the **action** being aborted must be individually TERMINATEd (see below).
TERMINATE	The TERMINATE operation is essentially equivalent to the LEAVE operation (i.e. it aborts an **action**) but it is not limited to specifying an enclosing action. In other words, TERMINATE can be used to abort any

concurrent activity. If the action being TERMINATEd is an enclosing action, this operation is identical to LEAVE. If the action to be TERMINATEd is not currently active, this operation is ignored. The main use of TERMINATE is to abort concurrent PROCESSes (See [8.3]).

RESTART — The RESTART operator is used to 'reset' an executing action. All actions initiated by the action to be restarted and not yet completed are terminated, as in the case of the LEAVE operation, before the action is restarted.

RESUME — The RESUME operator provides another mechanism to terminate the execution of an action. It differs from LEAVE in that LEAVE is followed by the label of the action to be terminated. RESUME is followed by the label of the action whose execution is to be continued. Any actions initiated during the execution of the action to be resumed and not yet completed are terminated.

Beware that these operators affect the sequence of operations and might be meaningless or unimplementable when used in parallel actions, e.g.:

Example: `X := (. . . NEXT . . . B=C; LEAVE X NEXT . . .)`

The example illustrates a possible source of ambiguety since no order of evaluation can be imposed on 'B = C ; LEAVE X'. When 'LEAVE X' is executed, the transfer 'B = C' may or may not have been completed.

5.5.3. Examples

Example:

```
I\Indirect<>      := VMA<13>,
X\IndeX<0:3>      := VMA<14:17>,
Y\Offset<0:17>    := VMA<18:35>,
VMA\Virtual.Memory.Address<13:35> :=                          ! PDP-10
        BEGIN
        REPEAT
                BEGIN
                IF X => Y = Reg[X] + Y NEXT       ! add the Index register
                DECODE I =>
                        BEGIN
                        0:= (VMA<13:17> = 0 NEXT LEAVE VMA),        ! done
                        1:= VMA = Memory[Y]<13:35>          ! Indirect loop
                        END
                END
        END
```

The body of the Virtual Memory computation specifies a (potentially infinite) loop of indirect address. Indexing through a register specified in the X field is performed (if X≠0) by adding the contents of the offset field to the register and truncating the result to 18 bits. After indexing has taken place, the indirect address field (1-bit) is tested. If the indirect bit is '0' the index field is cleared and the operation is completed (the effective address is left in its carrier, VMA). If the indirect bit is '1', the current value of the offset is used to access a memory location. The rightmost 23 bits of the word are loaded into the virtual memory address carrier and the operation is repeated from the start.

Example:

```
S(Key<0:3>)<> :=                ! Searches the first 512 words of Mp for KEY:
        BEGIN
        Index=0 NEXT
        REPEAT
                BEGIN
                IF MP[Index] EQL Key => (S = 1 NEXT LEAVE S) NEXT
                Index= Index+1 NEXT
                IF Index EQL 512 => (S = 0 NEXT LEAVE S)
                END
        END                                                    ! end of S
```

The search loop can be terminated under two conditions: (a) by finding a match or, (b) by exhausting the list. The carrier S<> is set to '1' or '0' respectively, to indicate the mode of termination. The carrier INDEX contains a pointer or address to the last location searched.

Example:

```
S(Key<0:3>)<> :=                ! Searches the first 512 words of Mp for KEY:
        BEGIN
        Index=0 NEXT
        S1:=    BEGIN
                IF Index EQL 512 => (S = 0 NEXT LEAVE S) NEXT
                IF MP[Index] NEQ Key =>
                        (Index = Index + 1 NEXT RESTART S1) NEXT
                S = 1
                END
        END                                                    ! end of S
```

The example is a variation on the table search of the previous example. Now however, the loop is built implicitly, by defining the body of the loop as a **labelled-action** and simply restarting it the inside.

Example:

```
Interpreter :=                                          ! Instruction Interpreter
        BEGIN
        . . . . . NEXT
        Icycle() NEXT                                ! Invoke the Instruction Cycle
        IF Error EQL 1 => BEGIN . . . . . END NEXT          ! Error Handler
        . . . . .
        END,
Icycle :=                                                ! Instruction Cycle
        BEGIN
        PC = PC + 2 NEXT                           ! Increment Program Counter
        IR = Rword(PC) NEXT                                ! Instruction Fetch
        DECODE IR<0:3> =>                                ! Operation Decoding
                BEGIN
                . . . . .
                ACC = ACC + Rword(IR<4:15>)                ! ADD Instruction
                . . . . .
                END,
Rword(Addr<0:11>)<0:15> :=                                ! Memory Access
        BEGIN
        IF Addr GTR Upper.Bound =>                        ! Boundary check
                (Error = 1 NEXT RESUME Interpreter) NEXT            ! Abort
        Rword = MP[Addr]                                   ! Memory Fetch
        END,
```

In the example, procedure INTERPRETER activates procedure ICYCLE which fetches, decodes, and executes the instructions. In doing so, ICYCLE activates procedure RWORD which is used to access the memory (MP) of the machine. RWORD checks that the memory address is in bounds before performing the access operation. If a boundary error is detected, a flag (ERROR) is set and the rest of the operation of ICYCLE is aborted (by returning to procedure INTERPRETER, at the point where it activated ICYCLE). It is up to the 'resumed' procedure (INTERPRETER) to take the proper corrective action, if any. Notice that we could have let ICYCLE handle the error by terminating RWORD with 'LEAVE RWORD'. However, this would have meant that the ICYCLE procedure had to check the error flag (ERROR) after every call to RWORD. Depending on the size or complexity of the description, this might be undesirable.

Example:

```
P(..)<..> :=
        Begin
        ..... Next
        L := ( A(..): B(x): C(..) Next ... ) Next
        .....
        End,

B(Y<0:2>) :=
        Begin
        Decode Y =>
                Begin
                '000 := Leave P,
                '001 := Restart P,
                '010 := Resume P,
                '011 := Terminate P,
                '100 := Terminate A,
                '101 := Leave B,
                '110 := Leave C,
                '111 := Leave L
                End
        End
```

The last example attempts to illustrate the full power and consequences of the use of the control operators of ISPS. An entity (P) contains a **labelled-action** (L), whose first step consists of concurrently invoking three other entities (A, B, and C). A and C are not really important and we will not specify their behavior. B however, takes one 'parameter' (Y) and depending on the value of the actual 'parameter' (X) it will select a particular control operation:

Value of X	Effect
0	The 'LEAVE P' operation aborts P and in the process, it also aborts A, B, C, and L.
1	The 'RESTART P' operation aborts A, B, C, and L, and execution continues from the beginning of P.
2	The 'RESUME P' operation aborts A, B, and C (not L). That is, the body of P (L) resumes execution.
3	The 'TERMINATE P' operation has the same effect as 'LEAVE P'.
4	The 'TERMINATE A' operation aborts A. B and C continue normally.

5 The 'LEAVE B' operation aborts B. A and C continue normally.

6 The 'LEAVE C' operation is in error. The activation of B was not nested inside the activation of C. Errors of this type are not always detectable by static analysis.

7 The 'LEAVE L' operation is in error. The label L is not accessible or know to B. Notice however, that if L also happens to be the name of an entity declared in the scope of B, the operation may or may not be valid, depending on the dynamic nesting of activations (in any event, the **identifier** L refers to the entity, not to the **labelled-action**).

5.5.4. Selecting the Right Operation

By a suitable rearrangement of the description, inserting or eliminating **labelled-actions**, etc, LEAVE, RESUME, and RESTART are more or less interchangeable[4]. It is a matter of style to select the best mechanism to describe the behavior of the computer. One must select whatever is more descriptive, clear, or in agreement with one's own personal bias.

Figure 5-1 contrasts the LEAVE, RESTART, and RESUME operations when used in static and dynamic contexts.

[4] They are all cases of 'return', LEAVE means 'return from ...', RESUME means 'return to ...', and RESTART means 'return to the head of ...'

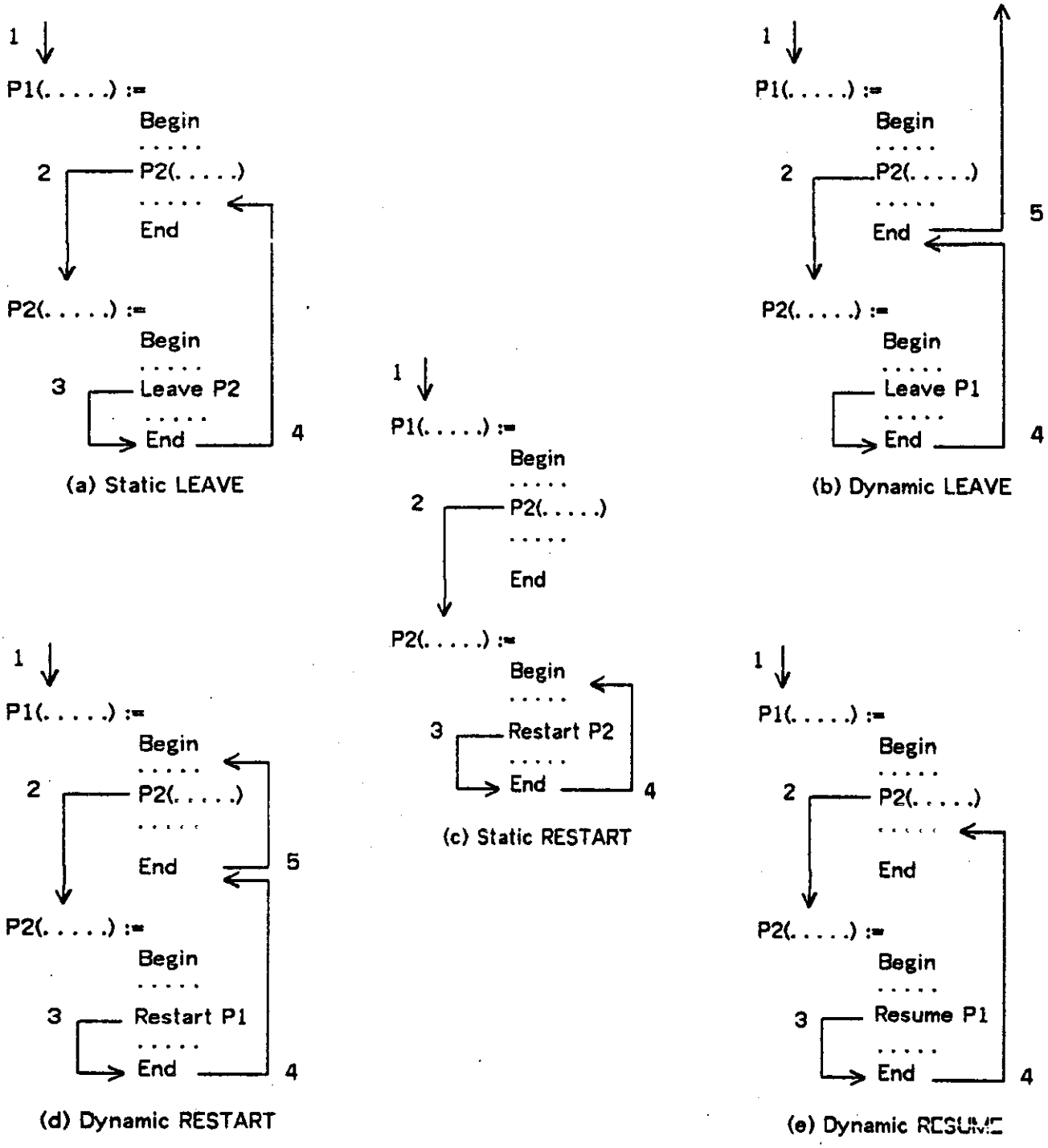


Figure 5-1: Static and Dynamic Use of LEAVE, RESTART, and RESUME

6. Carrier Expressions

```
c-expression ::=    c-disjunction | c-transfer
c-transfer ::=      e-access-LIST@-LIST^transfer-op transfer-op c-disjunction
                                                                  (See [7])

c-disjunction ::=   c-conjunction-LIST^or-op
c-conjunction ::=   c-relation-LIST^and-op
c-relation ::=      c-sum-LIST^rel-op
c-sum ::=           c-factor-LIST^add-op
c-factor ::=        c-shift-LIST^mult-op
c-shift ::=         c-concatenation-LIST^shift-op
c-concatenation ::= c-unary-LIST@
c-unary ::=         c-term | unary-op c-term                   (See [7])
```

Carrier expressions (c-expressions) describe a logical connection between carriers and operators. Each operation defines a carrier which can then be connected to other operators to define yet other carriers.

The syntax of a **c-expression** defines the precedence of the language operations (these are listed in Table [6-1], in increasing order of precedence; **unary-ops** have the highest precedence.) All operators in the same row have the same precedence and consecutive operations of the same precedence are executed from left to right, with the exception of the transfer operations which are executed from right to left.

```
transfer-op ::=   _ | = | <=
or-op ::=         OR | XOR
and-op ::=        AND | EQV
rel-op ::=        EQL | NEQ | LSS | LEQ | GTR | GEQ | TST
add-op ::=        + | -
mult-op ::=       * | / | MOD
shift-op ::=      SL0 | SL1 | SLR | SLO | SLI |
                  SR0 | SR1 | SRR | SRO | SRI

concat-op ::=     @
unary-op ::=      NOT | + | -
```

Table 6-1: Operator Precedence

Expression	Meaning
c-term	These are the basic carriers used to build expressions. Briefly, these include **constants**, entity carriers, **c-expressions** in parentheses, etc. See [7] for details.

c-unary	The result of applying a unary operator to a **c-term** carrier.
c-concatenation	The result of concatenating one or more **c-unary** carriers.
c-shift	The result of shifting or rotating **c-concatenation** carriers.
c-factor	The result of multiplying, dividing, etc. **c-shift** carriers.
c-sum	The result of adding or subtracting **c-factor** carriers.
c-relation	The result of applying a relational operator to **c-sum** carriers.
c-conjunction	The result of ANDing or EQVing **c-relation** carriers.
c-disjunction	The result of ORing or XORing **c-conjunction** carriers.
c-transfer	The result of transfering values from a **c-disjuntion** carrier to one or more concatenated **e-access** carriers. Multiple transfers from the same **c-disjunction** carrier can be specified .

6.1. Data Types and Arithmetic Representation

A bit pattern stored in a carrier has no semantic content. It is the context in which the carrier is used which determines whether the bit string is to be treated as a logical or arithmetic operand.

When treated as an arithmetic operand, a bit string can be interpreted in any of four standard arithmetic-representations: **Two's-Complement, One's-Complement, Signed-Magnitude,** and **Unsigned-Magnitude.** For the signed representations (all but the last), the sign bit is always the leftmost one.

By default, ISPS arithmetic operators assume a Two's Complement arithmetic representation for the values contained in the carriers. The mechanisms used to select a different arithmetic representation are described in [8.6].

When describing the logical and arithmetic operators, we will use the term 'sign-extension' to indicate the extension of an operand to match some length requirement. The meaning of 'sign extension' is dependent on the particular arithmetic representation used in the operation, as defined in [8.6].

The arithmetic representation also applies to **constants.** For binary, octal, and hexadecimal **constants,** the writer has explicit control over the bit patterns and can therefore specify positive or negative **constants.** Decimal **constants** are always positive (the 'sign bit' is always 0 as a consequence of the rules used to compute the length of decimal **constants.** See [3.3]).

6.2. Data Operators

6.2.1. Add-Op (Unary)

Unary + is treated as a no-op and is ignored by the parser.

Unary - defines a carrier whose value is the arithmetic complement of its operand. The length of the result is one bit longer that the length of the operand. This operation is invalid in unsigned arithmetic (negative numbers can not be represented).

Representation	Meaning
Signed Magnitude	In signed magnitude arithmetic, this operation simply inverts the sign bit (the leftmost bit of the result contains 0).
One's Complement	In one's complement arithmetic, this operation inverts every bit of the operand, including the sign bit (the leftmost bit of the result contains 0).
Two's Complement	In two's complement arithmetic, this operation inverts every bit of the operand, including the sign bit, and then increments the result by 1 (this is an unsigned addition). The leftmost bit of the result contains the carry out of this addition, if any.

6.2.2. NOT

The NOT operator defines a carrier whose length is equal to the length of its operand. NOT defines a carrier whose value is the bitwise logical complement of its operand.

6.2.3. @

The @ operator concatenates its left and right operands. The length of the result is the sum of the lengths of the operands.

6.2.4. Shift-Op

The shift operations perform a variable number of single bit shift and rotate operations on the left operand. The number of steps is determined by the value of the right operand or is implied to be 1 (SLI and SRI). The right operand is treated as an unsigned quantity. The length of the resulting carrier is the same as the length of the left operand.

The operator names indicate both the direction of shifting and the place were the input bits come from. All shift operators have a name of the form Sxy where x is either L(eft) or

R(ight) to indicate the direction of shifting, and y is either 0, 1, R, D, or I to indicate the source of shift-in bits. The first two (0 and 1) indicate a continuous stream of 0 or 1 bits, respectively. R indicates a Rotation i.e. the shift-out bits are routed back to the shift-in position. D indicates a Duplication, and the shift-in bits are simply a replication of the bit contained in the shift-in position. I indicates Immediate and the shift-in bit is the rightmost bit of the second operand (the SLI and SRI operations always shift the left operand 1 position to the left or right).

6.2.5. Mult-Op

The *, /, and MOD operators compute the arithmetic product, quotient, or remainder of their two operands.

The length of the result of the * operator is the sum of the lengths of its operands. The length of the result of the / operator is the length of the left operand. The length of the result of the MOD operator is the length of the right operand.

The sign of a product or quotient is computed according to the normal algebraic rules. The sign of the remainder (MOD operation) is the same as the dividend.

6.2.6. Add-Op (binary)

The binary + and - operators compute the arithmetic sum and difference of their two operands, respectively. These operations require that both operands be of equal lengths. If this is not the case, the shortest operand is sign-extended until it matches the length of the other operand. The length of the result in both cases is one bit longer than the longest operand.

In Unsigned, Two's Complement and One's Complement addition (subtraction), the extra bit added to the length of the result contains the carry (borrow) out of the most significant bit position.

In Signed Magnitude arithmetic, the sign bit is not treated as part of the number (as in the complement representations) and the most significant bit of the operands is the bit to the right of the sign. The result of a signed magnitude addition (subtraction) has the following format: 1) the carry (borrow) out of the most significant position occupies the leftmost bit of the result, 2) the sign of the result appears in the second bit from the left, 3) the rest of the result. In other words, all representations use the extra bit added to the length of the result to store the carry/borrow out of the operation.

6.2.7. Rel-op

The EQL, NEQ, LSS, LEQ, GTR, GEQ, and TST operators perform an arithmetic test between their operands. The length of the result of the TST operation is two bits long and encodes a FORTRAN-like IF statement (i.e. the result is either '00, '01, or '10 depending on whether the left operand is less than, equal to, or greater than, the right operand). All other relational operators produce 1 bit long results, indicating whether the relation is True ('1) of False ('0).

As with the + and - operators, these operators require that their operands be of the same length. This is accomplished by sign-extending the shortest operand until it matches the length of the longest operand. The relational operations are performed in the context of a specific arithmetic representation, thus, positive and negative zero (possible in sign magnitude and one's complement representations) are EQuaL.

6.2.8. And-Op

The AND and EQV operators perform the bitwise conjunction (logical product) and coincidence operations on their operands, respectively. The length of the result is equal to the length of their operands. If the operands are not of the same length, the shortest operand is expanded by concatenating enough 0 bits on its left until it matches the length of the longest operand.

6.2.9. Or-Op

The OR and XOR operations perform the bitwise disjunction (logical sum) and exclusive-or operations on their operands, respectively. The length of the result is equal to the length of their operands. If the operands are not of the same length, the shortest operand is expanded by concatenating enough 0 bits on its left until it matches the length of the longest operand.

6.3. Transfer Operation

The _, =, and <= operators are used to transmit values between carriers. The _ and = operators are equivalent and perform a logical-transfer (the right operand is extended with zeroes or is truncated on the left if the lengths of the operands are not equal).

The <= operator performs an arithmetic-transfer (the right operand is sign-extended until it matches the length of the left operand. If the right operand is longer that the left operand, a truncation will occur).

For completeness, the 'result' of a transfer operation is the result of the evaluation of the

right operand (before any truncation or extension takes place).

Although the **transfer-ops** play a role similar to that of an assignment operator in a programming language, an important difference must be understood: A transfer operator is simply a connection between carriers, it is the nature of the carriers that determines the implementation of the connection. Thus, syntactically, there is no difference between a gated carrier transfer (writing a value into a memory) and the loading of a bus (making a value available to some lines.)

6.3.1. Storing into Concatenated Carriers

The transfer operators are different from the other operators in that they take a value from a carrier and place it into another. Because of this difference, only one certain operator is allowed on the left side of a transfer. This operator is '@' or concatenation. Since the operator @ is defined to be a carrier 'grouping' operator it defines carriers which might be read from as well as written into.

```
A@B=D                                   ! Valid Transfer
A+B=C                                   ! Invalid Transfer
```

The first example takes the D carrier and 'splits' it between A and B. D is zero-expanded if A@B is larger. The leftmost bit of (the expanded) D is loaded into the leftmost bit of A; the rightmost bit of D is loaded into the rightmost bit of B. The second example (A+B=D) is invalid. It implies that the right hand side carrier (D) is loaded into the left hand side carrier which happens to be the output of an adder.

6.3.2. Multiple Transfers

As indicated in the syntax, multiple transfers of an expression can be described:

```
R1 = R2[X] <= R3 = R4@R5 = A + B
```

which is equivalent to:

```
Temporary = A + B next
R1 = Temporary;
R2[X] <= Temporary;
R3 = Temporary;
R4@R5 = Temporary;
```

A multiple transfer implies a 'broadcast' of the rightmost carrier. Truncations or extensions take place on each individual transfer.

7. Carrier Terms

```
c-term: : =          e-access |                                        (See [3.3])
                     constant |                                        (See [3.7])
                     constant < name-pair > |                            (See [6])
                     constant < c-expression > |
                     ( c-expression ) |
                     ( c-expression ) < name-pair > |
                     ( c-expression ) < c-expression > |
e-access: : =        identifier ac-set as-set                          (See [3.2])
ac-set: : =          nil | () | (c-expression-LIST,)
as-set: : =          word-as-set bit-as-set
word-as-set: : =     nil | [ c-expression ]
bit-as-set: : =      nil | < name-pair > | < c-expression >
```

A carrier term (**c-term**) defines the primitive operands that are used to build **c-expressions**.

Carrier Term	Description
e-access	An entity access (**e-access**) serves two roles. It is used to connect (a portion of) the carrier associated with an entity to the data operators. It is also used to activate the body of an entity. The context and format of the **e-access** determines which of these roles it is playing (this is explained in the following sections).
	identifier — The **identifier** is used to uniquely select the entity.
	ac-set — The actual connection set (**ac-set**) defines a set of carriers to match the carriers specified in the formal connection set (**fc-set**) of the entity declaration.
	as-set — The actual structure set (**as-set**) defines a subset of the formal structure set (**fs-set**) specified in the entity declaration.
	word-as-set — The word structure selects one of the 'words' specified in the declaration of the entity.
	bit-as-set — The bit structure selects one or more consecutive bits specified in the declaration of the entity.
constant	A **constant** defines a carrier whose value can not be modified. The structure of a **constant** is defined by its base and length (See [3.3]). **Constant** carriers are assumed to have the structure <N:0>, where N+1 is the length of the constant. One or more bits of a **constant** can be accessed by specifying the bit names inside '<' and '>'.
c-expression	A **c-expression** can be enclosed in parenthesis, to specify an order of evaluation different from that which is implied by the precedence of the

operators. A carrier defined by a **c-expressions** in parenthesis has a default bit-naming convention, <N:0>, where N+1 is the length of the carrier. One or more bits of the carrier can be selected by enclosing the bit names in '<' and '>'.

As described in [4.2], the general format of an **e-head** consists of an **identifier**, , a list of interface carriers (optional), a word structure (optional), and a bit structure (optional). Depending on which of the optional parts were specified in the **e-head**, the entity can be read, written, or activated.

An **e-access** also consists of an **identifier**, a list of interface carriers (optional), a word structure (optional), and a bit structure (optional). Depending on which of the optional parts were specified in the **e-access**, the entity will be read, written, or activated.

7.1. Read/Write Access

It is possible to read from or write into the carrier associated with an entity without activating it; all that is needed is to specify the entity name and a structure. If only the name is used, then the entire carrier, as defined in the entity declaration, is used.

An actual structure set (**as-set**) is used to specify a subset of the formal structure set (**fs-set**) used in the declaration of the entity's carrier.

Example: `... = .. + A[x]<6:7> ...`

A **c-expression** used to select a word or a bit of a carrier must evaluate to a value corresponding to one of the words or bits named in the entity declaration. Notice that while it is valid to access any number of consecutive bits, only one word can be specified (i.e. no word **name-pairs** are allowed in an access).

7.2. Activation

It is possible to activate an entity without accessing its carrier; all that is needed is to specify the entity name and a connection.

An actual connection set (**ac-set**) consists of a list of carriers that are connected to, or transferred into the corresponding elements of the formal connection set (**fc-set**, see [4.2]). For details on the actual mechanism used to establish the connection between actual and formal carriers see [8.2].

Example: `... NEXT X(Y<3:4>); Z() NEXT ...`

7.3. Combined Access and Activation

It is possible to both access and activate an entity; all that is needed is to specify the entity name, a structure, and a connection. An **e-access** can appear, as a **c-term** , on either side of a **transfer-operator**. If no activation is implied (i.e. there is no **ac-set**), then the carrier is simply read or written, depending on whether is it on the right or left of the **transfer-operator**, respectively. If an activation is implied (i.e. there is an **ac-set**), then the body is activated before (after) the carrier is read (written).

Example: `... X(Y)<6:7> ...`

7.4. Compatibility Between Use and Declaration

A valid **e-access** must be compatible with the declaration of the entity:

1. A valid **e-access** can specify a word structure (**word-as-set**) if and only if the entity declaration specified a word structure (**word-fs-set**).

2. A valid **e-access** can specify a bit structure (**bit-as-set**) only if the entity declaration specified a bit structure (**bit-fs-set**). If no bit structure is specified in the access, the full bit structure used in the declaration is assumed.

3. A valid **e-access** can specify a list of interface carriers (**ac-set**) if and only if the entity declaration specified a list of interface carriers (**fc-set**) or an **e-body**. An empty list ('()') must be used to activate an entity which does not have any formal interface carriers.

7.5. Examples

```
(Breg + Ireg + Displacement) <23:0>
```

The result of the additions is truncated by taking the rightmost 24 bits.

```
Mode @ #177777<15:0>
```

Although '#177777' is an 18-bit constant (6 octal digits), the use of '<15:0>' in fact defines a 16-bit **constant** carrier.

```
ACC<0> = X<B<3:0>>
```

The example shows the use of a **c-expression** to select an arbitrary bit in a carrier. The contents of B<3:0> determines a bit name. Since this value can range from 0 to 15, X must

have been declared as a carrier of the form X<n:0> or X<0:n> where n ≥ 15.

```
X(R)<5> = . . .
```

1) Compute the expression to the right of the '=' operator, 2) Take the rightmost bit of the result and store it into bit 5 of X, 3) Connect (or copy) R to the formal interface carrier of X, and finally, 4) Activate X.

```
X = Z()
```

1) Activate Z (Z does not have any interface carriers), 2) When the activation is completed take the value in the carrier Z<..> and store it into X<..>. If Z has the PROCESS attribute (See [8.3]) the transfer takes place immediately, at the start of the activation (i.e. the value in the Z carrier may be ambiguous).

```
VMA = IR<13:35> NEXT EA = VMA()<18:35> NEXT
```

1) Load the virtual memory address carrier (VMA) 2) Activate the entity (VMA), 3) Load the effective address carrier (EA).

```
VMA() = IR<13:35> NEXT EA = VMA<18:35> NEXT
```

performs the same sequence of operations as the previous example.

8. Predefined Qualifiers

The previous chapters have provided the basic syntax and semantics for the declaration of entities, and their carriers, interfaces, and bodies. In this chapter we define means for extending the basic semantics of declarations. These extensions are based on the general qualifier mechanism, to be described in [9].

8.1. INCREMENT Qualifier

The increment attribute changes the interpretation of the word names used in the left hand side of a mapped array declaration. Normally, there are as many words as there are integers in the **name-pair** used to declare the word structure (**word-fs-set**) . When the INCREMENT attribute is used, the actual number of words is obtained by dividing the size of the range by a user specified INCREMENT value:

e-head { INCREMENT : **constant** } := **e-head**

The word names in the range are obtained by dividing the original word names by the increment value (integer division) and then multiplying the quotient by the increment value.

There are some limitations in the use of the INCREMENT attribute:

1. The INCREMENT attribute is only valid (or meaningful) when used in mapped array declarations in which the 'word size' of the left hand side is a multiple of the 'word size' on the right hand side.

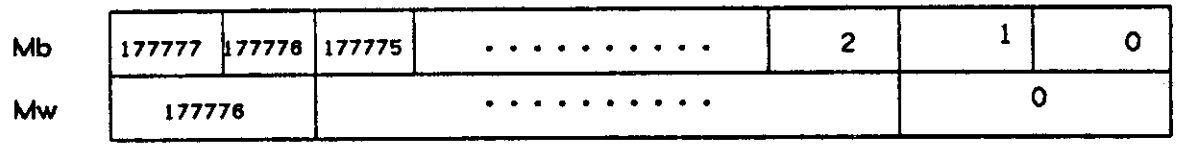
2. If one member of an array mapping chain is qualified by INCREMENT, ALL members of the chain must map onto arrays of shorter words.

3. A chain of mappings propagates the INCREMENT attributes in such a way that an array mapped onto another has an increment attribute which is the product of the increment attributes towards the base of the chain.

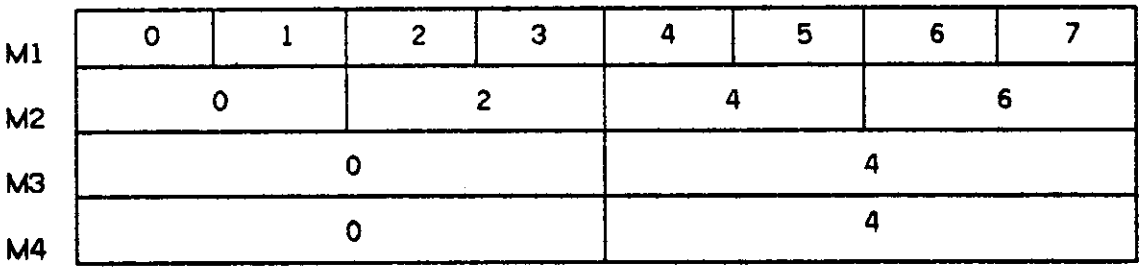
Example:

```
Mb[#177777:0]<7:0>,                                          ! PDP-11 Byte Memory
Mw[#177777:0]<15:0> {INCREMENT:2} := Mb[#177777:0]<7:0>      ! Word Memory
```

In the example above, Mw is defined to have 64K/2 words, named #177776, #177774,....#4,#2,#0. Each word maps exactly over two consecutive bytes. Moreover, notice that the mapping also specifies that the even-address byte contains the low-order bits of a word, as defined in the PDP-11 architecture. Pictorially, the equivalence of the address spaces is shown in Figure 8-1 (a).



(a) PDP-11 Address Space



(b) Multiple Mappings

Figure 8-1: INCREMENT Qualifier

Structure Mappings do not have to be organized as a linear chain of declarations, as the following example shows:

Example:

```
m1[0:7]<0:7>,
m2[0:7]<0:15> {increment:2} := m1[0:7]<0:7>,        chain is M2,M1
m3[0:7]<0:31> {increment:2} := m2[0:7]<0:15>,             M3,M2,M1
m4[0:7]<0:31> {increment:4} := m1[0:7]<0:7>,                M4,M1
```

The increment attribute attached to M3 is 2*2=4. That is, M3's addresses increment twice as fast as M2's addresses and these, in turn increment twice as fast as M1's addresses. M4 has an increment attribute of 4; it is not affected by M2's increment because M2 is not in its chain of mappings. There are 8/2=4 elements in M2 (M2[0], M2[2], M2[4], and M2[6]) and 8/4=2 elements in M3 (M3[0] and M3[4]). M3[4], M3[5], M3[6], and M3[7] refer to the same memory location: M3[4], which in turns maps onto M1[4:7]. Pictorially, this is represented in Figure 8-1 (b).

8.2. REFerence Qualifier

The default implementation of an interface (**fc-set**, See [4.2]) is by means of storage units which are loaded when the entity is activated (See [7.2, 7.3]).

The default mechanism for establishing a connection between the actual and the formal interface carriers in an entity activation is by copying the values contained in the actual carriers into the formal carriers, zero-extending or truncating on the left if necessary. The mechanism can be though of as a carrier transfer of the form:

```
Formal = Actual
```

In some applications the default interface mechanism outlined above might be too limiting and ISPS provides an alternative mechanism to specify the implementation of the interface carrier and the connection mechanism:

```
REF e-head
e-head { REF }
```

When a formal carrier is qualified with the string 'REF', the interface is not a storage unit, local to the entity, but instead it is a REFerence to some external carrier, to be specified at the activation site.

Example: `F(REF Reg<0:7>) := Begin ..... End`

When F is activated, no transfer of data between the actual carrier and the formal carrier (REG) takes place. REG is simply 'connected' to whatever carrier was specified at the call site. This connection remains in effect throughout the length of the activation. The mechanism can be though of as a carrier mapping of the form:

```
formal := actual
```

The mapping is established at the time of the connection and is subject to the same limitations normal mappings have (i.e. no truncation or extension is permitted).

An entity can have both REFerence and local interface carriers, as suggested in Figure 8-2. (a) and (b) display two alternative interfaces. The former uses local carriers for all the input operands while the latter uses REFerence carriers for the data operands while using a local carrier to retain the value of the third operand (the function code). (c) and (d) display the connection mechanisms. The former indicates that the local carriers are loaded

instantaneously, at the onset of the activation (this is suggested by the 'strobe' signal used to load the carriers). The latter indicates that the REFerence carriers are connected (in both directions) to the actual operand carriers throughout the length of the activation (this is suggested by the 'level' signal used to connect the carriers).

Actual carriers that correspond to REF formal carriers must be **e-accesses** and not arbitrary **c-expressions**. This is because, in principle, a REF formal carrier can be read or written.

The mapping of a REF carrier applies to both the structure of the actual carriers and to the behavioral part, it any. That is, the entity being activated can read/write/activate the actual entities through the formal carrier name.

Although the syntax allows otherwise, the **ac-set** of an actual **e-access** involved in a REF connection must be empty (i.e. '()').

Example:

```
P1(X<0:3>, REF Y<0:1>) := BEGIN . . . . . END,
R2<0:1>,
P2 := BEGIN . . . NEXT P1('C,R2) NEXT . . . . END
```

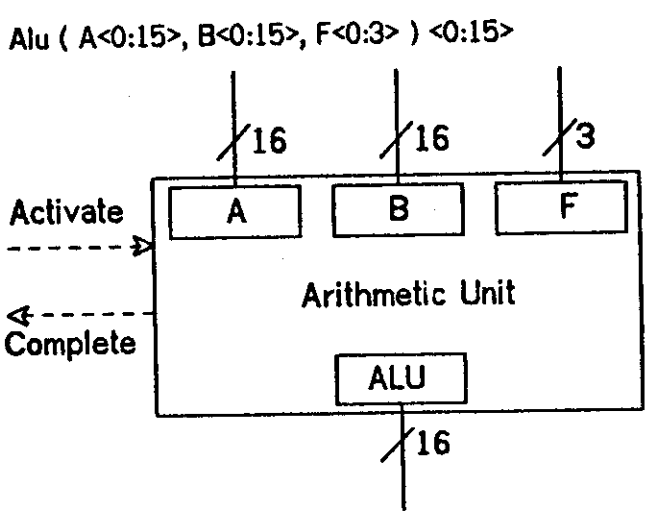
In the above example, P1 requires two interface carriers, one of which (Y) has the REF attribute. The activation of P1 inside P2 copies the **constant** 'C (decimal 12) into X (a register local to P1) and maps Y onto R2 ('Y<0:1> := R2<0:1>'). During the activation of P1, R2 is accessible and known as Y. P1 can read and/or write Y (i.e. R2).

The use of REF connections can be used to describe complex behaviors:

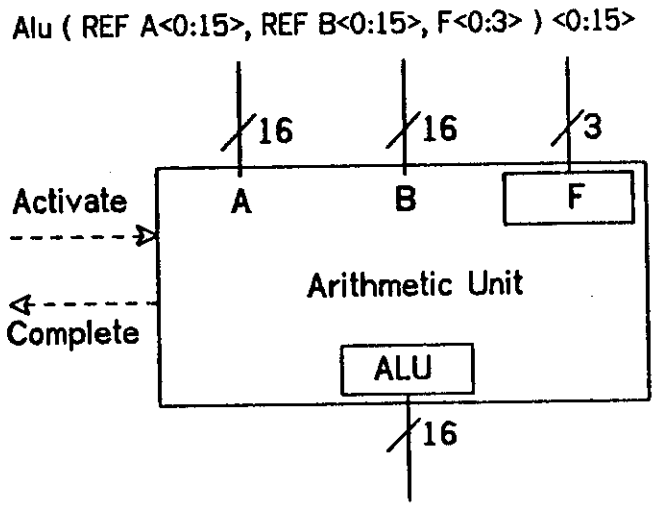
Example:

```
P1(REF X()<0:1>) := BEGIN . . . . . END,
P2(REF Y()<0:1>) := BEGIN . . . . . END,
A()<0:1> := BEGIN . . . . END,
P3 := BEGIN . . . NEXT P1(A); P2(A) NEXT . . . END,
```

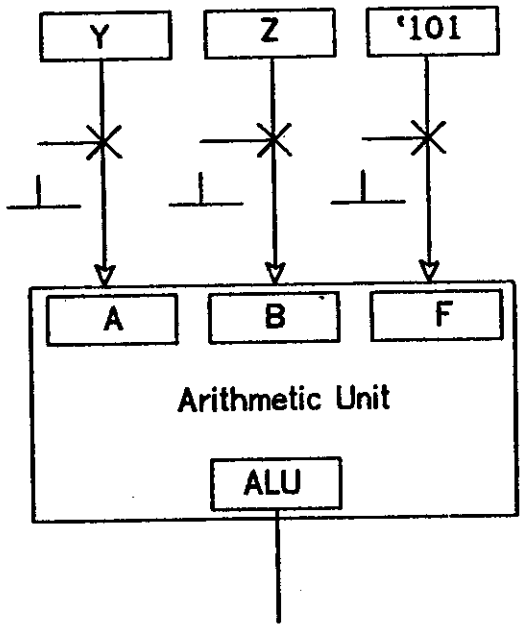
In the example, when P3 activates both P1 and P2, it connects to both of them the entity (A). Since both P1 and P2 can read/write/activate A, they can affect each other's behavior.



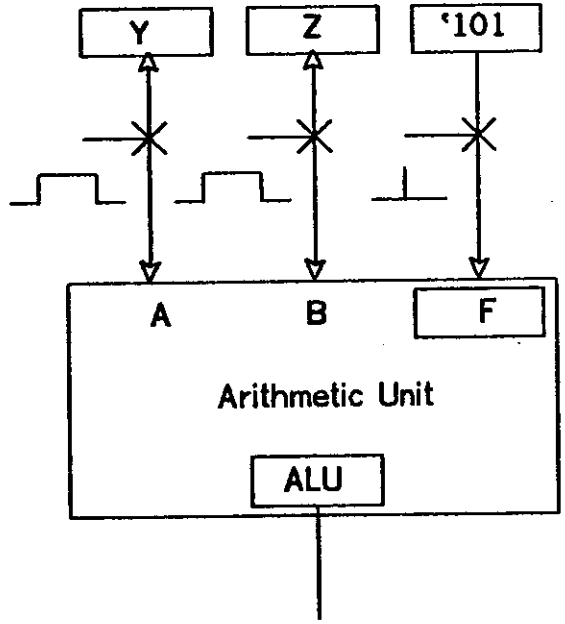
(a) Local Interface Carriers



(b) REFerence Interface Carriers



(c) Interface Transfers



(d) REFerence Interface Connections

Figure 8-2: REFerence Qualifier

8.3. PROCESS and CRITICAL Qualifiers

By default, entities are activated as if they were procedures or functions in a programming language. That is, the activating entity waits until the activated entity completes its operation. Although this mechanism is a useful abstraction, it is not powerful enough to describe the behavior of complex hardware units. An entity declaration can specify a departure from the default by means of the PROCESS and CRITICAL qualifiers:

```
PROCESS e-head := e-body
e-head { PROCESS} := e-body
CRITICAL e-head := e-body
e-head { CRITICAL} := e-body
```

The qualifier PROCESS can be used as a attribute (prefix or inside '{}') to a declaration to indicate that the entity is to be executed as an asynchronous control environment. An activation of an entity marked as 'process' results in the creation of a control 'token' for the entity. It then starts executing concurrently with the caller.

```
Example:  PROCESS ALU(A<0:15>, B<0:15>, F<0:3>)<0:15> := ... ,
```

The qualifier CRITICAL can be used as a attribute (prefix or inside '{}') to a declaration to indicate that the entity contains an arbitration mechanism so that one and only one activation of the entity can be in progress at the same time. Activations are queued if the entity is already active.

```
Example:  CRITICAL arbiter := Begin ..... End,
```

PROCESS and CRITICAL are independent attributes. The former controls the continuation of the callers, the latter controls concurrent callers. When both qualifiers are present, CRITICAL takes precedence. That is, the caller of a CRITICAL PROCESS entity is delayed until the entity is free or idle before it continues executing in parallel.

Figure 8-3 suggests the control of activations in the presence of the PROCESS and CRITICAL attributes.

Attempting to activate an already active, non-critical entity is an error and it yields unpredictable results.

The arbitration mechanism implied by the CRITICAL qualifier applies not only to the activation of the entity but also to the evaluation and connection of interface carriers. This is to avoid conflicting use of the formals (REF or otherwise). Thus, the callers wait until the entity has completed its current activation before establishing a new connection.

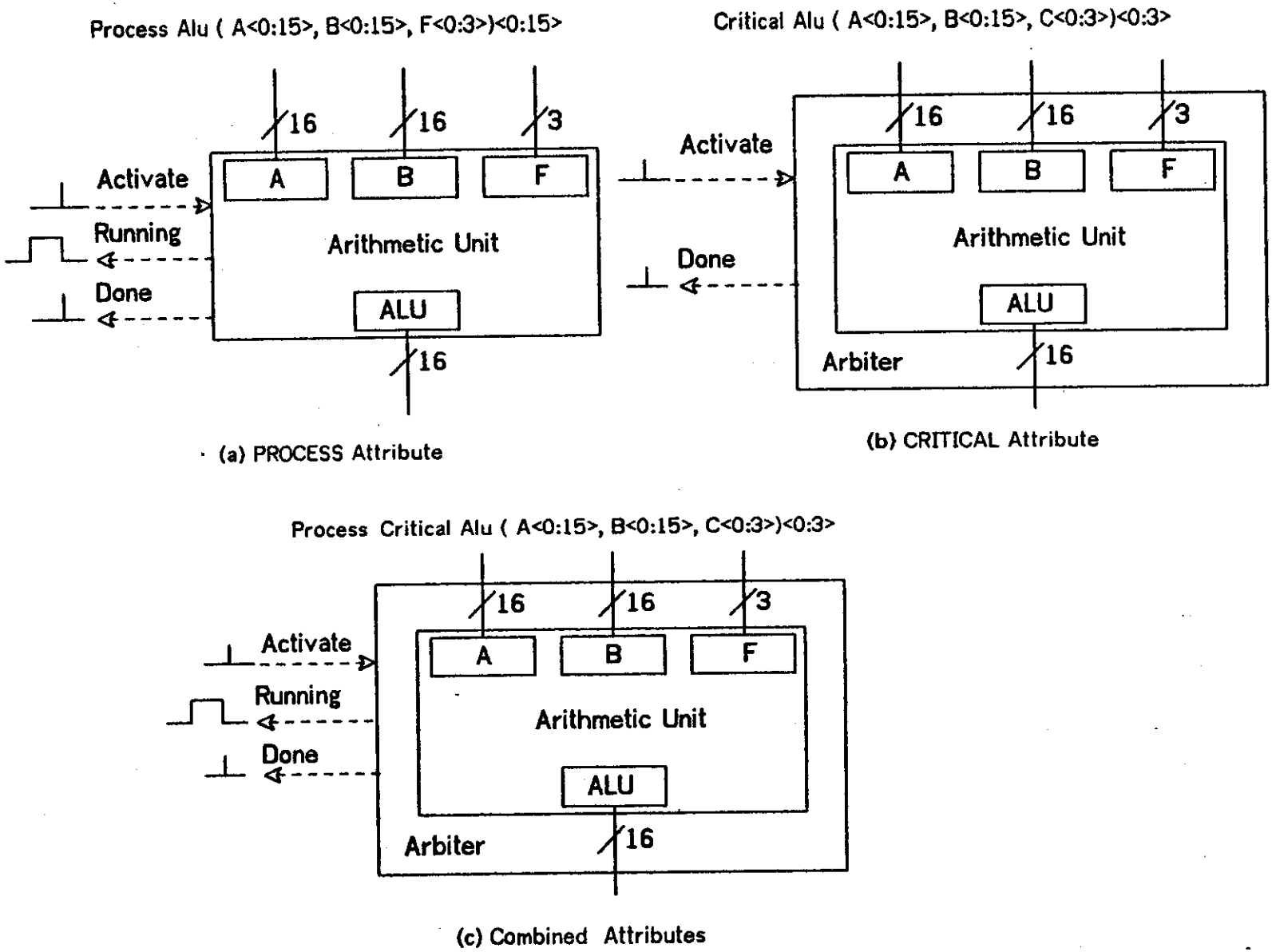


Figure 8-3: PROCESS and CRITICAL Qualifiers

8.4. MAIN Qualifier

As seen in [4.3], an **e-body** can consist of a list of **sections**, each of which can contain a list of entity declarations (**e-declarations**). The qualifier MAIN is used to identify the 'main' entry point of an entity or ISPS description:

```
MAIN e-head := e-body
e-head { MAIN } := e-body
```

MAIN serves to identify which one of the internal entities is to be executed when the enclosing entity is activated. This applies to either the entire ISPS program or an internal declaration, to any level of nesting.

Example:

```
P1 (...) :=
        Begin
        ** ... **
        P2 (...) := Begin .... End,
        P3 (...) := Begin .... End,
        MAIN  P4 := Begin .... End,
        P5 (...) := Begin .... End
        End
```

The effect of invoking P1 is equivalent to invoking its main internal entity, P4, directly (of course, since that entity is internal, it is not directly available to the caller.)

When activating an entity with an internal, main entity, the connection is done through the interface carriers, if any, of the enclosing entity.

> Although syntactically correct, the presence of an interface (**fc-set**) or carrier (**fs-set**) in the declaration of the main internal entity is useless since there is no way for a caller to access these internal carriers.

Figure 8-4 depicts the activation of a complex entity, containing several internal entities, one of which has the MAIN attribute.

8.5. PTIME Qualifier

The PTIME qualifier is used to specify the average or expected 'execution' time for the body of an entity. This is useful in some applications such as simulation, synthesis, or verification. The qualifier is specified as an attribute of the declaration of the entity:

e-head { PTIME : **constant** } := **e-body**

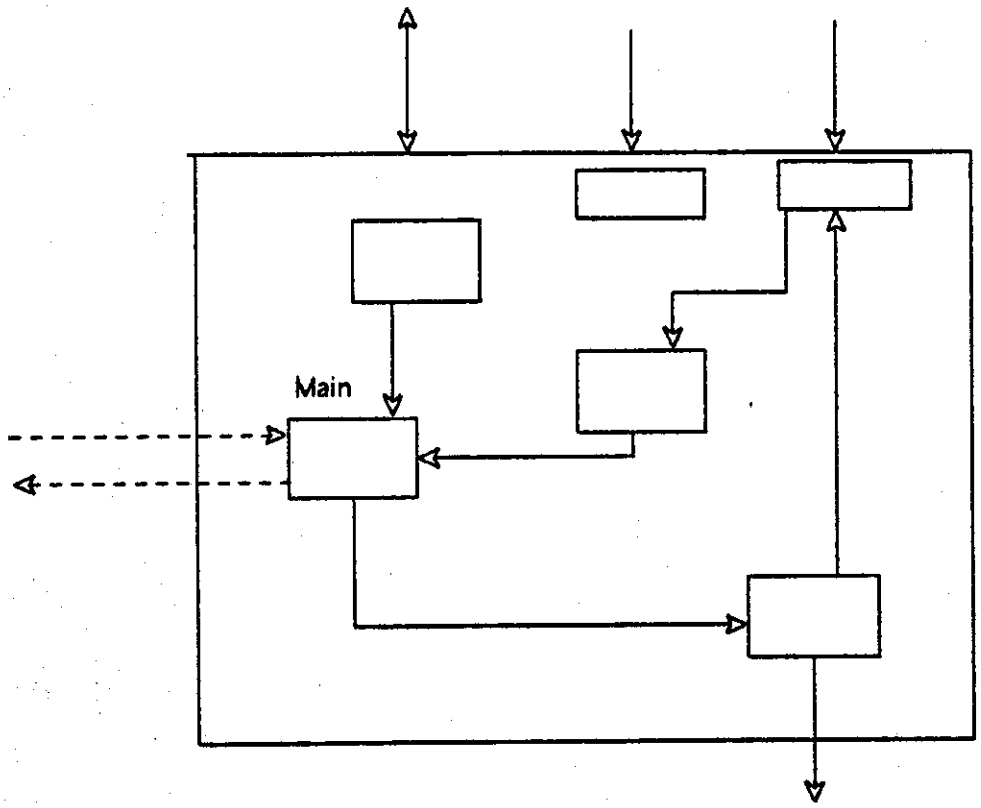


Figure 8-4: Use of the MAIN Qualifier

The value of the qualifier is a constant specifying the number of 'time units' needed for the average execution of the entity body.

Example: `P(A<,,>,B<,,>) {PTIME:25} := Begin ..... End`

This mechanism for specifying timing is only an approximation. If more detailed timing is desired, the body of the entity can specify via DELAY, WAIT, and TIME.WAIT (See [11]) the actual time needed for each control path.

8.6. Arithmetic Qualifiers

The selection of the representation to be used in the context of an arithmetic operation is indicated by one of the following qualifiers:

Qualifier	Meaning
{TC}	used to indicate that the operation is to be done in Two's Complement arithmetic. If an operand needs to be sign-extended, the extension is done by replicating the sign bit.
{OC}	used to indicate that the operation is to be done in One's Complement arithmetic. If an operand needs to be sign-extended, the extension is done by replicating the sign bit.
{SM}	used to indicate that the operation is to be done in Signed Magnitude arithmetic. If an operand needs to be sign-extended, the eshxtension is done by inserting '0' bits between the sign bit and the rest.
{US}	used to indicate that the operation is to be done in Unsigned arithmetic. If an operand needs to be sign-extended, the extension is done by adding '0' bits to the left of the carrier.

The selection of arithmetic representation can be done over a single operation (attaching the modifier to the operator), over an entire **b-expression** (attaching the modifier to one of the BEGIN/END brackets), or finally, over an entire **section** (attaching the modifier to the **section-header**. The following example will illustrate this:

Example:

```
Sample :=
        BEGIN                   ! By default, all arithmetic is Two's Complement
        ** Section.1 **
        . . . . .
        ** Section.2 ** (OC)                    !  Section is One's Complement
        . . . . .
        F(X<0:5>) := BEGIN (SM) . . . . . END       ! F is Signed Magnitude
        . . . . .
        G(Y<0:5>) :=                                ! G is One's Complement
                BEGIN
                . . . . .
                IF  Y<0>  =>  BEGIN  (UM)  . . . . . END
              ! The conditional action uses Unsigned Magnitude as default.
                . . . . .
                Y  = Y +  (TC) 2  NEXT          ! Two's Complement Addition
                . . . . .
                END

        ** Section.3 **                         ! Return to Two's Complement
        . . . . .
        END
```

9. Qualifiers

In this chapter we specify the mechanism for specifying information of interest to an application program. In general, the parser will only perform syntactic checks on these constructs since it has no means to ascertain their semantic correctness. The mechanism (Qualifiers) has already been introduced in previous sections (e.g. REF, INCREMENT, etc.), now we give the formal specification.

```
q-set ::=          { q-av-pair-LIST; }
q-av-pair ::=      identifier |                          (See [3.2])
                   identifier : |
                   identifier : q-value-LIST,
q-value ::=        identifier |
                   constant |                            (See [3.3])
                   quoted-text |                         (See [3.6])
                   q-set
```

The qualifier set (**q-set**) is used to specify lists of attribute/value pairs which are used to extend the semantics of an ISPS description.

Example:

```
ALU(F<0:3>,A<0:15>,B<0:15>)<0:15>{SPEED:250; MODULE:SN74181}:=.....
```

The syntax of a **q-set** indicates that in some instances an **identifier** can stand for both the attribute name and the list of values. This is allowed to simplify the writting of certain qualifier values that uniquely identify the attribute (If one wishes to be explicit about it, '**identifier:**' can be used instead. The ':' indicates that the preceding **identifier** is an attribute name). Notice that qualifiers can be arbitrarily nested (i.e. a **q-set** is a valid **q-value**.)

9.1. Placement of Qualifiers

Qualifiers can appear in several contexts in a description:

1. After an **e-head**, following the **identifier**, **fc-set**, or **fs-set**, whichever is the last component of the **e-head** (before the ':=', if present). This applies to **e-heads** in any context: declarations, mappings, and formal carriers.

2. After the brackets ('BEGIN', 'END', '(', and ')') used to enclose an **e-body** or a **block-action**.

3. After the **identifier** of a **labelled-action**, before the ':=' operator.

4. After the 'IF' and 'DECODE' keywords in a **conditional-action**, before the **c-expression**.

5. After any data operator, including arithmetic, logical, transfer, etc., before the rigth operand.

6. After an **e-access**, following the **identifier**, **ac-set**, **as-set**, whichever is the last component of the **e-access**.

9.2. Identifier Sequences

For convenience and readability, it is sometimes necessary to display qualifiers in a context that can be easily noted. This can be achieved with **id-sequences** of the form:

id-sequence : : = **identifier** | (See [3.2])
identifier id-sequence

The ISPS parser will treat all **identifiers** preceding the last **identifier** of the sequence (if any) as qualifiers. These qualifiers are lumped together with whatever qualifiers were explicitly defined inside '{}', if any (notice that these **identifiers** stand for both the attribute name and the value list as indicated above). An **id-sequence** can appear anywhere an **identifier** is valid. Thus, the following are equivalent:

Examples:

```
A[0:255]<0:3> {ROM: CONNECT:LINK2},
ROM A[0:255]<0:3> {CONNECT: LINK2},
```

9.3. Summary of Predefined Qualifiers in ISPS

An initial set of qualifiers has been predefined in the language. These qualifiers have already been introduced in this document. Here we simply list them, indicating their format. For additional details about each of these qualifiers, the reader must consult the sections were these qualifier are introduced.

Qualifier	Format	Usage
TC	K,Q	Two's Complement Arithmetic
OC	K,Q	One's Complement Arithmetic
SM	K,Q	Signed-Magnitude Arithmetic
US	K,Q	Unsigned Arithmetic
INCREMENT:n	Q	Structure Mapping
PTIME:n	Q	Execution Time
CRITICAL	K,Q	Protected Activities
PROCESS	K,Q	Independent Activities
REF	K,Q	Interface Carriers

The column labelled Format indicates wheather the attribute can be used as a Keyword preceding an entity name (i.e. as part of a id-sequence) or as a Qualifier, enclosed in '{}'.

10. Other Declarations

other-declarations ::=	REQUIRE.ISP **quoted-text** |	(See [3.6])
	MACRO **identifier m-parameter-set := quoted-text** |	(See [3.2])
	ISPS-definition	
m-parameter-set ::=	nil |	
	() |	
	(identifier-LIST)	
ISPS-definition ::=	DEFINE **identifier := q-set** |	(See [9])
	DEFINE **identifier := quoted-text** |	
	DEFINE **identifier := constant**	(See [3.3])

10.1. REQUIRE

The reserved keyword REQUIRE.ISP is used to signal the expansion of a an external file inside the ISPS description. The **quoted-text** describes the file name. The expansion takes place at the point the REQUIRE.ISP construct appears:

Example: **REQUIRE.ISP | MARK1.ISP[L410MB25] |,**

10.2. MACRO

The reserved keyword MACRO provides a simple mechanism to declare text strings that are to be substituted for instances of the **identifier** in the ISPS description. Optional parameters can be specified by enclosing a list of **identifiers** inside parenthesis. These 'formal parameters' are matched by corresponding 'actual parameters' at the expansion site. The actual parameters can be any 'expression' in ISPS. The use of any type of brackets, ('(', ')', '[', ']', etc.) in an actual parameter is permitted, provided its partner is also part of the actual parameter.

Example:

```
MACRO t1(l,body) :=                    ! Two parameters: Length and Body
        |                                              ! Macro delimiter
        Begin
        ** s **                              ! create a dummy section name
        temp<0:l>,                  ! create a temp  of the right length.
        Main t := Begin body End                ! create a 'procedure'
        End
        |,                                             ! Macro delimiter

p2 := t1(7,.....)                    ! This use of the macro expands to:

p2 :=   Begin
        ** s **
        temp<0:7>,
        Main t :=
                Begin
                .....
                End
        End
```

10.3. DEFINE

The reserved keyword DEFINE is used to name a q-set, a constant, or a quoted-text.

Examples:

```
Define ROM := {MODULE: SN74187; SPEED: 40},
Define MSIZE := 255,
```

11. Predeclared Entities

The following entities are predeclared in the language:

COUNT.ONE(..)<..> is a predeclared entity which has a structure, and whose activation COUNT.ONE(expression) returns the number of non-zero bits in the expression. The length of the result is equal to the decimal value of the length of the expression, regardless of the value of the expression[5]. For instance, if the expression is 16 bits long, the result of COUNT.ONE is ALWAYS 6 bits long (5 bits to express 16 plus a leading 0).

DELAY(..) is a predeclared entity which does not have a structure and whose invocation, DELAY(expression), does not have side effects. DELAY terminates its activation after a number of application-defined time units given by the value of the expression.

FIRST.ONE(..)<..> is a predeclared entity which has a structure, and whose invocation FIRST.ONE(expression) returns the number of leading zeros in the value of the expression (i.e. the number of zeros before the first one, hence the name). If the expression is all zeroes, the result is the length of the expression. The length of the result follows the rule defined for COUNT.ONE.

IS.RUNNING(..)<> is a predefined entity which has a 1-bit structure and whose invocation IS.RUNNING(entity.name) returns 1 (true) if entity.name is currently active, 0 (false) otherwise.

LAST.ONE(..)<..> is a predeclared entity which has a structure, and whose activation LAST.ONE(expression) returns the number of trailing zeroes in the value of the expression (i.e. the number of zeroes after the last one). The length of the result is identical to that of FIRST (COUNT) .ONE.

MASK.LEFT(..,..)<..> is a predeclared entity which has a structure, and whose activation MASK.LEFT(expr1,expr2) returns a result with the same length as EXPR1. The leading EXPR2 bits are set to 0, the remaining bits retain the value they had in EXPR1. Basically, this function builds a mask of LENGTH(EXPR1) bits with EXPR2 bits on the left set to 0 and the rest set to 1. It then computes its result by ANDing the mask with EXPR1. If EXPR2 is equal to 0 (unsigned comparison), the result is identical to EXPR1. If EXPR2 is greater than the length of EXPR1, the result is all 0s.

MASK.RIGHT(..,..)<..> is a predeclared entity which has a structure, and whose activation MASK.RIGHT(expr1,expr2) is identical to MASK.LEFT but cleans up the bits on the right of EXPR1 using EXPR2 to compute the number of bits.

NO.OP() is a predeclared entity which does not have a structure and whose behavior has no side effects. NO.OP() can be used as a null action.

---

[5] Whenever the expression 'decimal value' is used it means a number whose length follows the rules of ISPS for decimal numbers, that is, a number whose length is exactly one bit longer than the smallest number of bits needed to represent the number. This is to avoid problems when performed signed arithmetic. Decimal numbers are ALWAYS positive since their leading bit is 0!

PARITY(..)<> is a predeclared entity which has a 1-bit structure and whose activation PARITY(expression) returns the odd-parity bit of the expression (it is equivalent to COUNT.ONE(EXPRESSION) MOD{US} 2).

STOP() is a predeclared entity which does not have a structure and whose invocation, STOP(), terminates the activation of ALL entities.

TIME.WAIT(..,..)<..> is a predeclared entity which has a structure and whose invocation TIME.WAIT(expr1,expr2) combines the effect of the WAIT and DELAY entities. TIME.WAIT continuously evaluates EXPR1 until it is non-zero or until the number of time units represented by EXPR2 has been exceeded. EXPR2 is computed exactly once, at the beginning of TIME.WAIT. When the activation is completed, TIME.WAIT returns the final value of EXPR1 (the length of the result is the same as the length of EXPR1). Depending on the value returned, the caller can decide whether EXPR1 yielded a non-zero value or the time-out limit provided by EXPR2 was exceeded before EXPR1 became non-zero.

UNDEFINED()<..> is a predeclared entity which has some structure and whose activation UNDEFINED() returns a carrier of undetermined length, whose value is unknown. Activations of UNDEFINED are guaranteed to terminate after some undetermined amount of time.

UNPREDICTABLE() is a predeclared entity which does not have a structure and which exhibits a totally unpredictable behavior. It is different from UNDEFINED() in that the latter preserves the flow of control. An activation of UNPREDICTABLE() is not guaranteed to terminate or that upon termination, control will return to the activation site.

WAIT(..)<..> is a predeclared entity which has a structure and whose invocation, WAIT(expression), continuously evaluates the expression. WAIT terminates its activation when the value of the expression is not equal to 0. WAIT returns the last value of the expression (the non-zero value which terminated the activation; the result has the same length as the expression).

12. Reserved Keywords and Identifiers in ISPS

AND	Logical Operator
CRITICAL	only when used as qualifier
DECODE	Conditional action Selector
DEFINE	Special Declaration
EQL	Arithmetic Operator
EQV	Logical Operator
GEQ	Relational (Arithmetic) Operator
GTR	Relational (Arithmetic) Operator
IF	Conditional action Selector
INCREMENT	only when used as qualifier
K	only when attached to a constant
LEAVE	Control Operator
LEQ	Relational (Arithmetic) Operator
LSS	Relational (Arithmetic) Operator
M	only when attached to a constant
MACRO	Special Declaration
MOD	Arithmetic Operator
NEQ	Relational (Arithmetic) Operator
NEXT	Sequencing Operator
NOT	Logical Operator
OC	only when used as qualifier
OR	Logical Operator
PROCESS	only when used as qualifier
PTIME	only when used as qualifier
REF	only when used as qualifier
REQUIRE_ISP	Special Declaration
REPEAT	Control Operator
RESTART	Control Operator
RESUME	Control Operator
SL0	Shift Operator
SL1	Shift Operator
SLD	Shift Operator
SLI	Shift Operator
SLR	Shift Operator
SM	only when used as qualifier
SR0	Shift Operator
SR1	Shift Operator
SRD	Shift Operator
SRI	Shift Operator
SRR	Shift Operator
TC	only when used as qualifier
TERMINATE	Control Operator
TST	Relational (Arithmetic) Operator
US	only when used as qualifier
XOR	Logical Operator

13. Using the ISPS Parser

The following reproduction of a session running the parser should be self explanatory. The parser accepts the specifications of a source file (ISPS) and produces a listing file (optional) and an object file containing the parse tree. There are several switches that can be appended to the input file specification. These switches control several options with regard to the generation of the listing and the object file.

```
.run isps
ISPS Translator V5B(1)-7
(/H for Help)

*/h
 File specifications follow normal CUSP convention:

      <object>.GDB,<listing>.LST=<source>.ISP/<switch>/<switch>/...

 Abbreviated file specifications can be used:
      FilNam.ISP      ; GDB has same name as ISP.  No Listing file.
      FilNam.ISP/L    ; GDB and LST have same name as ISP.

 A switch is  turned on by '/<letter>' and  off by '/-<letter>':
 (* indicates a switch turned on by default)
 A : All.  Same as switch combination /E/L/O/R/S/W.
 C : Comments.  Comments from ISP are placed in the GDB file.
 E : Expand.  Macro invocations are printed in the LST file.
 H : Help.  This text.
 K : Keep.  Macro definitions will appear in the GDB file.
 L : List.  Causes a listing file to be generated.
 N : No program.  LST file will only contain error messages.
 O*: Original.  GDB file will have ISP format numbers
 P : Position.  Put out Line/Page position on terminals.
 R*: Readable.  GDB file has lexeme names instead of numbers.
 S : Symbols.  Print out name of declarations as they are parsed.
 W : Watch.  Prints a skeleton trace of compiler phases as they occur.
 X : SyntaX.  Quick syntax-only check of ISP.  No GDB file.

*test

 TEST.ISP  1.

      Errors:        0
      Warnings:      0
      Stack Used:    74 of 512 Words
      PStack Used:   13 of 200 Words
      Max. Core Used: 3 + 18 K
      Machine Time:  00:00:00
      People Time:   00:00:00

EXIT
```

The above procedure will create a file TEST.GDB which contains the object program (parse tree). No listing was requested and none was generated. If the parser detects errors in the source program appropriate messages are typed on the user's terminal. If a listing file is being generated, the error messages will also appear in the listing.

14. ISPS Global Data Base: File Format and Syntax

Floret Silva Nobilis
Floribus et Foliis

Carl Orff, *Carmina Burana*, 1937

The purpose of the Global Data Base (GDB) is to provide a means of representing a machine description in a manner that is amenable to manipulation by many programming languages. This is accomplished by storing the GDB as an ASCII file with a specific format. Any language which can read a file as a stream of characters can work with the GDB.

A GDB file contains a representation of the parse tree of an ISPS description. When the description is processed and transformed into a GDB representation, the entire information content is retained. Because of this the transformation is reversible. A GDB may be changed back into an ISPS description with 'ease'.

14.1. GDB Header Line

The first line of a GDB file contains a header which gives information about the format of the rest of the file, the compiler version, the source file name, and the date and time of compilation. The rest of the file (from the second line on) is the tree representation of an ISPS description.

A typical header line looks like:

```
GDB:A;ISPS Compiler V5B-7;DSK:TEST.ISP[N655MB25];17 Jun 79;23:18:25;
```

The character following 'GDB:' is a letter which indicates the form of the information in the tree. Currently there are four formats, 'A', 'B', 'C', and 'D'. Each of the formats uses the same syntax, only the printing-form of the information is different. The differences will be discussed later.

14.2. The GDB Syntax

A non-terminal of the language appears as a sub-tree within the parse tree. The syntax for a sub-tree is an open parentheses ("(") followed by a node-name representing the non-terminal. This is in turn followed by a (possibly empty) list of sons of the non-terminal after which comes a close parentheses (")"). i.e.

```
( <non-terminal.node.name> <son1> <son2> .... <sonN> )
```

The sons of the non-terminal may be non-terminals themselves or they may be terminals

(leaves of the tree). A terminal appears in the tree as a string of characters, without enclosing parenthesis. Both non-terminal node-names and terminals can be followed by a list of "attributes". The general form of an attribute is

```
!xx!yyyyy!
```

The string 'xx' is a decimal integer indicating the type of the attribute. (The types will be discussed later, in [14.4]) The string 'yyyyy' is an arbitrary character string where occurrences of '!' are represented as '!!'. Thus, in general, a GDB node looks like:

```
( <node-name> <attribute1> ... <attributen>
        <son1> <attribute1> .... <attributej>
        . . . . . .
        <sonn> <attribute1> .... <attributek> )
```

Spaces, Tabs, Carriage-Returns, and Line-Feeds appear in the GDB trees only as delimiters between elements. (Except for the interior of quoted-texts and node attributes which may have arbitrary characters.) Multiple occurrences of any or all of these delimiter characters are considered equivalent to a single occurrence of any of them.

Different types of non-terminals have different numbers of sons. However each type of non-terminal has a particular number of sons and the position and type of the sons is fixed. In the tree, all of the sons of a node which are not Nil (LISP style Nil) actually appear in the tree. Nodes which are Nil only appear if some son appears after them which is not Nil. An example will serve to demonstrate. An E-Access[6] has five sons. The E-Access "A{QUALITY}" appears in the tree as:

```
(EACCESS A NIL NIL NIL (QSET QUALITY))
```

The E-Access "A(3)" appears in the tree as:

```
(EACCESS A (ACSET 3))
```

The Nil sons on the 'end' of the E-Access were left out of the tree.

---

[6]Whenever possible, node-names have been derived from the production names used in the BNF description of ISPS

14.3. Representation of Node-Names and Terminals

There are two representations for node-names in the GDB file. Which one is used is controlled by a compilation switch (See [13]). The R switch controls the representation of the node-names.

By default, node-names appear as an ASCII string which has some mnemonic relation to the production name used in the BNF. The list of node-names is described in [15].

If the -R (complement of R) switch is used during the compilation, the GDB node-names will appear as octal numbers. The equivalence between these numbers and the node names is subject to change without notice. If you feel that you need the table, contact the maintainers.

The following example will help clarify the difference between these two formats:

```
test<0:77> :=
        BEGIN
        test _ 123456789123456789 next
        test _ 0 next
        test _ Not test
        END
```

The above ISPS file, when compiled with the default setting of the R switch, produces the following GDB file:

```
GDB:A;ISPS Compiler V5B-7:DSK:TEST.ISP[N655MB25]:17 Jun 79:23:18:25:
(ISPSDECLARATION
   (EDECLR
      (EHEAD TEST NIL NIL (: 0 77))
      (NEXT
         (_ (EACCESS TEST) 123456789123456789)
         (_ (EACCESS TEST) 0)
         (_ (EACCESS TEST) (NOT (EACCESS TEST))))))
```

When the same ISPS file is compiled using the -R switch, the GDB file looks like this:

```
GDB:C;ISPS Compiler V5B-7;DSK:TEST,ISP[N655MB25];17 Jun 79;23:20:47;
(1
   (2
      (6 TEST NIL NIL (40 0 77))
      (23
         (117 (171 TEST) 123456789123456789)
         (117 (171 TEST) 0)
         (117 (171 TEST) (170 (171 TEST))))))
```

There are three types of terminals that can appear in a GDB tree. They correspond to **identifiers** , **constants** , and **quoted-text** strings. **identifiers** and **quoted-text** strings appear in the GDB tree exactly as they appear in the original ISPS source file (with the exception of lower case letters which are mapped into upper case.)

There are two representations for **constants** . The compiler switch O is used to select which format is to be generated.

By default, **constants** in a GDB file appear exactly as they appear in the original ISPS source file (See [3.3]). The above examples used this format.

If the -O (complement of O) switch is used during the compilation, **constants** have the following format in the GDB tree:

```
#xxxx<yy>
```

The string 'xxxx' is an octal number which is the value of the corresponding ISPS **constant.** The string 'yy' is a decimal number which indicates the EXACT bit length of the octal **constant.**

The following GDB file was generated from the above ISPS source file, using the -O switch:

```
GDB:B;ISPS Compiler V5B-7;DSK:TEST,ISP[N655MB25];17 Jun 79;23:21:50;
(ISPSDECLARATION
   (EDECLR
      (EHEAD TEST NIL NIL (: #0<2> #115<8>))
      (NEXT
         (_ (EACCESS TEST) #6664664565464057425<58>)
         (_ (EACCESS TEST) #0<2>)
         (_ (EACCESS TEST) (NOT (EACCESS TEST))))))
```

The following table describes the relation between the switch settings and the GDB format

generated:

Format	R	0	
A	/R	/0	(default)
B	/R	/-0	
C	/-R	/0	
D	/-R	/-0	

14.4. Attribute Types

A node-name or terminal may have any number of attributes following it. The format of an attribute is

```
!xx!yyyyy!
```

The string 'xx' is a decimal number indicating the type of the attribute and the string 'yyyyy' is the body of the attribute. The body may contain any ASCII character except NULL (0) and occurrences of '!' are represented by '!!'.

The attribute types are:

1. This attribute corresponds to a **comment** which appeared in the ISPS description. The comment appeared at the end of the line which this node was located on. This type of attribute appears in the tree only if the /C switch is used in the compiler.

2. This attribute corresponds to a blank line which appears in the ISPS description. This only appears if the /C switch is used in the compiler.

3. This attribute corresponds to an ISPS **alias**. If an alias is given to a **constant** or to an **identifier** in ISPS this attribute will appear in the tree, following the **constant** or **identifier**.

4. This attribute corresponds to a **comment** which appears at the end of the ISPS description. It only appears in the tree if the /C switch is used in the compiler.

5. This attribute provides the line and page number in the ISPS description that a node appeared on. This is only given for terminals (**constants** and **identifiers**). These attributes appear in the tree only if the /P switch is used in the compiler. The attribute looks like !4!500/4! for line 500 on page 4. The numbers are decimal.

6. This attribute contains the name of a labeled block, e.g.: "Begin |name| .... End |name|" produces an attribute of "!5!name!".

15. GDB Node Types

For the rest of this document we will assume that the reader knows the grammar of ISPS.

Not all of the non-terminals in the BNF are used in the tree. A great effort has been made to remove any nonessential redundancy from the tree. For instance, the simple ISPS register access of "A" is parsed as a cunary which has a definition of a **c-term** which is defined as **e-access** which is defined as **identifier** which is "A". The tree for this would look something like:

```
(CUNARY (CTERM (EACCESS A)))
```

The **cunary** and the **c-term** are unnecessary and the GDB file is simplified to:

```
(EACCESS A)
```

The following is a list of the valid node types followed by their representation in the tree. The upper case **identifiers** are the literal strings (node-names) which appear in the tree. The lower case strings are subtree-type names. For example, 'c-expression' may be replaced by any valid tree which corresponds to an ISPS c-expression parse tree. These include the sub-trees whose roots are '+', '*', 'SHO', 'MOD', 'EACCESS', etc.

The form used in the tables below is:

```
ISPS-BNF-Name\GDB-File-Print-Name(s)
        <first tree representation>
        <second tree representation>
                . . . .
        <last tree representation>

Discussion.
```

The tree representations given are those that appear in the tree if all sons indicated so are not NIL. The actual node in the tree need only have as many sons as are not NIL. If a NIL son is followed by a non-NIL son then both will appear in the tree. (See the examples in [14]).

In the tree representations, any representation which is not contained in '()' indicates the name of a sub-tree which can appear in place of the node type under discussion.

```
ISPS-Declaration\ISPSDECLARATION
        (ISPSDECLARATION e-declaration)
```

The node ISPSDECLARATION is always the root of the parse tree. It appears once, as the first node of the GDB file.

```
E-Declaration\EDECLR
        (EDECLR e-head e-body)
        (XDEFINE identifier q-set q-set)
        (XDEFINE identifier quotedtext q-set)
        (XDEFINE identifier constant q-set)
        (XMACRO identifier fcset quotedtext)
        e-head
```

An EDECLR node appears only if an ISPS entity declaration has an **e-body**.

Macros are expanded during the compilation and then, usually, thrown away. The /K switch forces the compiler to keep the macro declaration in the tree. The FCSET son of a XMACRO node is a special case of the general FCSET node. The elements of the FCSET are identifiers, not arbitrary EHEADs.

```
E-Head\EHEAD
        (EHEAD identifier fc-set word-fs-set bit-fs-set q-set)
```

The q-set of an EHEAD node is specified in the ISPS description as a set of keywords preceding the identifier, or as a set of qualifiers inside { and } after the **e-head** (before the ':=' or ',', as the case may be.)

```
E-Body\EBODY
        (EBODY s-action q-set)
        (EBODY section-list q-set)
        s-action
        section-list
        e-head
```

The node EBODY will not actually appear in the tree unless the Q-Set is non-NIL. The q-set of an EBODY node is specified in the ISPS description as a set of qualifiers inside { and } after the BEGIN or END brackets surrounding the **e-body**.

```
FC-Set\FCSET
        (FCSET)
        (FCSET e-head .... e-head )
        NIL
```

The FC-Set node with no sons (' (FCSET) ') appears only when a '()' appeared in the ISPS description.

```
Word-FS-Set\[f]
        name-pair
        NIL
```

The Word-FS-Set node never appears in the tree.

```
Bit-FS-Set\<f>
        (<f>)
        name-pair
        NIL
```

The Bit-FS-Set node with no sons (' (<f>) ') appears only when a '<>' appeared in the ISPS description.

```
Name-Pair\:
        (: constant constant)
        constant

Q-Set\QSET
        (QSET q-av-pair .... q-av-pair )
        NIL

Q-AV-Pair\:q:
        (:q: identifier q-value-list)
        identifier

Q-Value-List\,q,
        (,q, q-value .... q-value )
        q-value

Q-Value\QVALUE
        identifier
        constant
        quotedtext
        q-set
```

The Q-Value node never appears in the tree.

```
Section-List\SECTIONLIST
        (SECTIONLIST section .... section)
        section
```

The Section-List node appears in the tree only if there is more than one section.

```
Section\SECTION
        (SECTION identifier e-declaration-list q-set)
```

The q-set of a SECTION node is specified in the ISPS description as a set of qualifiers inside { and } following the closing '**' of the section-header.

```
E-Declaration-List\EDECLRLIST
        (EDECLRLIST e-declaration .... e-declaration)
        e-declaration
        NIL
```

The E-Declaration-List node appears in the tree only if there is more than one E-Declaration.

```
S-Action\NEXT
        (NEXT p-action .... p-action)
        p-action
```

The S-Action node appears in the tree only if there is more than one P-Action.

```
P-Action\;
        (; action .... action)
        action
```

The P-Action node appears in the tree only if there is more than one Action.

```
Action\ACTION
        block-action
        labelled-action
        c-expression
        conditional-execution
        conditional-decode
        control-action
```

The Action node never appears in the tree.

```
Block-Action\BLOCKACTION
        (BLOCKACTION s-action q-set)
        s-action
```

The Block-Action node appears in the tree only if there is a non-NIL Q-Set. The q-set of a BLOCKACTION node is specified in the ISPS description as a set of qualifiers inside { and } following the BEGIN or END surrounding the action.

```
Labelled-Action\LABELLEDACTION
        (LABELLEDACTION identifier action q-set)
```

The q-set of a LABELLEDACTION node is specified in the ISPS description as a set of qualifiers inside { and } following the identifier, before the ':='.

```
Conditional-Execution\IF
        (IF c-expression action q-set)
```

The q-set of an IF node is specified in the ISPS description as a set of qualifiers inside { and } following the IF operator.

```
Conditional-Decode\DECODE
        (DECODE c-expression numbered-list q-set)
```

The q-set of an DECODE node is specified in the ISPS description as a set of qualifiers inside { and } following the DECODE operator.

```
Numbered-List\NUMBEREDLIST
        (NUMBEREDLIST numbered-action .... numbered-action)

Numbered-Action\:=n
        (:=n constant action)
        (:=n name-pair action)
        (:=n name-list action)
        (:=n otherwise action)
        action

Otherwise\OTHERWISE
        (OTHERWISE)

Name-List\,n,
        (,n, name-pair .... name-pair)
        name-pair
```

The Name-List node appears in the tree only if there is more than one name-pair.

```
Control-Action\LEAVE, RESTART, RESUME, TERMINATE, REPEAT
        (LEAVE identifier)
        (RESTART identifier)
        (RESUME identifier)
        (TERMINATE identifier)
        (REPEAT action)
```

```
C-Expression\CEXPRESSION
        c-transfer
        c-disjunction
        c-conjunction
        c-relation
        c-sum
        c-factor
        c-shift
        c-concatenation
        c-unary
```

The C-Expression node never appears in the tree.

```
C-Transfer\_, <=
        (_ c-expression c-expression q-set)
        (<= c-expression c-expression q-set)
```

See note after NOT node

```
C-Disjunction\OR, XOR
        (OR c-expression c-expression q-set)
        (XOR c-expression c-expression q-set)
```

See note after NOT node

```
C-Conjunction\AND, EQV
        (AND c-expression c-expression q-set)
        (EQV c-expression c-expression q-set)
```

See note after NOT node

```
C-Relation\EQL, NEQ, LEQ, GEQ, LSS, GTR, TST
        (EQL c-expression c-expression q-set)
        (NEQ c-expression c-expression q-set)
        (LEQ c-expression c-expression q-set)
        (GEQ c-expression c-expression q-set)
        (LSS c-expression c-expression q-set)
        (GTR c-expression c-expression q-set)
        (TST c-expression c-expression q-set)
```

See note after NOT node

```
C-Sum\+, -
        (+ c-expression c-expression q-set)
        (- c-expression c-expression q-set)
```

See note after NOT node

```
C-Factor\*, /, MOD
        (* c-expression c-expression q-set)
        (/ c-expression c-expression q-set)
        (MOD c-expression c-expression q-set)
```

See note after NOT node

```
C-Shift\SR0, SR1, SRD, SRR, SRI, SL0, SL1, SLD, SLR, SLI
        (SR0 c-expression c-expression q-set)
        (SR1 c-expression c-expression q-set)
        (SRD c-expression c-expression q-set)
        (SRR c-expression c-expression q-set)
        (SRI c-expression c-expression q-set)
        (SL0 c-expression c-expression q-set)
        (SL1 c-expression c-expression q-set)
        (SLD c-expression c-expression q-set)
        (SLR c-expression c-expression q-set)
        (SLI c-expression c-expression q-set)
```

See note after NOT node

```
C-Concatenation\@
        (@ c-expression c-expression q-set)
```

See note after NOT node

```
C-Unary\CUNARY
        c-term
        c-negation
        c-complement
```

The C-Unary node never appears in the tree.

```
C-Negation\++, --
        (++ c-expression q-set)
        (-- c-expression q-set)
```

The unary-plus nodes are all thrown away. Only unary-minus nodes appear in the tree. See note after NOT node.

```
C-Complement\NOT
        (NOT c-expression q-set)
```

The q-set in an operator node is specified in the ISPS description as a set of qualifiers inside { and } following the operator.

```
C-Term\CTERM
        (CTERM constant bit-as-set)
        (CTERM c-expression bit-as-set)
        e-access
        constant
        c-expression

E-Access\EACCESS
        (EACCESS identifier ac-set word-as-set bit-as-set q-set)
```

The q-set of an EACCESS node is specified in the ISPS description as a list of keywords preceding the bnf or as a set of qualifiers inside { and } following the **e-access**.

```
Word-As-Set\[a]
        c-expression
```

The Word-AS-Set node never appears in the tree.

```
Bit-As-Set\<a>
        access-pair
```

The Bit-As-Set node never appears in the tree.

```
Access-Pair\:a:
        (:a: constant constant)
        c-expression

AC-Set\ACSET
        (ACSET)
        (ACSET c-expression .... c-expression)
        NIL
```

The ACSET node with no sons (' (ACSET) ') appears only when a '()' appeared in the ISPS description.

16. A Complete GDB Example

The following is a listing of the entire GDB file generated from the MARK1 ISPS Description.

16.1. ISPS Description

```
!                The Manchester University Mark-1 Computer
!
!This is the ISPS description of the first version of the machine, as
!reported in [Lavington, S.H. "A History of Manchester Computers",
!National Computing Centre Publications, Manchester, England, 1975]
!
!                    Mario R. Barbacci (BARBACCI@CMUA)
!
MARK1 :=
        Begin
        ** Memory.State **
        M[0:8191]<31:0>,

        ** Processor.State **
        PI\Present.Instruction<15:0>,
                F\Function<0:2> := PI<15:13>,
                S<0:12> := PI<12:0>,
        CR\Control.Register<12:0>,
        Acc\Accumulator<31:0>,

        ** Instruction.Execution ** (TC)
Main I.Cycle :=
        Begin
        PI = M[CR]<15:0> next
        Decode F =>
                Begin
        0\JMP   := CR = M[S],
        1\JRP   := CR = CR + M[S],
        2\LDN   := Acc = - M[S],
        3\STO   := M[S] = Acc,
        4:5\SUB := Acc = Acc - M[S],
        6\CMP   := If Acc Lss 0 => CR = CR + 1,
        7\STP   := Stop(),
                End next
        CR = CR + 1 next
        Restart I.Cycle
        End
End
```

16.2. GDB File

```
GDB:A;ISPS Compiler V58-7;DSK:MARK1.ISP[N655M825];18 Jun 79;00:12:12;
(ISPSDECLARATION
   (EDECLR
      (EHEAD MARK1)
      (SECTIONLIST
         (SECTION MEMORY.STATE (EHEAD M NIL (: 0 8191) (: 31 0)))
         (SECTION
            PROCESSOR.STATE
            (EDECLRLIST
               (EHEAD PI !2!PRESENT.INSTRUCTION! NIL NIL (: 15 0))
               (EDECLR
                  (EHEAD F !2!FUNCTION! NIL NIL (: 0 2))
                  (EHEAD PI NIL NIL (: 15 13)))
               (EDECLR
                  (EHEAD S NIL NIL (: 0 12))
                  (EHEAD PI NIL NIL (: 12 0)))
               (EHEAD CR !2!CONTROL.REGISTER! NIL NIL (: 12 0))
               (EHEAD ACC !2!ACCUMULATOR! NIL NIL (: 31 0))))
         (SECTION
            INSTRUCTION.EXECUTION
            (EDECLR
               (EHEAD I.CYCLE NIL NIL NIL (QSET MAIN))
               (NEXT
                  (_ (EACCESS PI) (EACCESS M NIL (EACCESS CR) (:a: 15 0)))
                  (DECODE
                     (EACCESS F)
                     (NUMBEREDLIST
                        (:=n
                           0 !2!JMP!
                           (_ (EACCESS CR) (EACCESS M NIL (EACCESS S))))
                        (:=n
                           1 !2!JRP!
                           (_
                              (EACCESS CR)
                              (+ (EACCESS CR) (EACCESS M NIL (EACCESS S)))))
                        (:=n
                           2 !2!LDN!
                           (_
                              (EACCESS ACC)
                              (-- (EACCESS M NIL (EACCESS S)))))
                        (:=n
                           3 !2!STO!
                           (_ (EACCESS M NIL (EACCESS S)) (EACCESS ACC)))
                        (:=n
                           (: 4 5 !2!SUB!)
                           (_
                              (EACCESS ACC)
                              (- (EACCESS ACC) (EACCESS M NIL (EACCESS S)))))
                        (:=n
                           6 !2!CMP!
                           (IF
                              (LSS (EACCESS ACC) 0)
                              (_ (EACCESS CR) (+ (EACCESS CR) 1))))
                        (:=n 7 !2!STP! (EACCESS STOP (ACSET)))))
                  (_ (EACCESS CR) (+ (EACCESS CR) 1))
                  (RESTART I.CYCLE)))
            (QSET TC)))))
```

17. References

[Barbacci,1978] M.R. Barbacci: "An Introduction to ISPS". Technical Report, Department of Computer Science, Carnegie-Mellon University, 1978. Report CMU-CS-78-137.

[Barbacci,1979] M.R. Barbacci: "Instruction Set Processor Specifications (ISPS): The Notation and its Applications". Technical Report, Department of Computer Science, Carnegie-Mellon University, 1979. Report CMU-CS-79-123.

[Bell,1971] C.G. Bell and A. Newell: Computer Structures: Readings and Examples, Mc-Graw Hill Book Company, New York, 1971.

[Bell,1978] C.G. Bell, J.C. Mudge, and J.E. McNamara: Computer Engineering, A DEC View of Hardware Systems Design. Digital Press, 1978.

Appendix I
Syntax Charts

This appendix contains the complete syntax of ISPS. It is presented in a pictorial format. All productions of the form X-LIST^{Y} are explicitly defined. The charts show explicitly all the places were a qualifier can appear.

Whenever the keywords 'BEGIN' and 'END' appear, they can be replaced by '(' and ')' respectively.

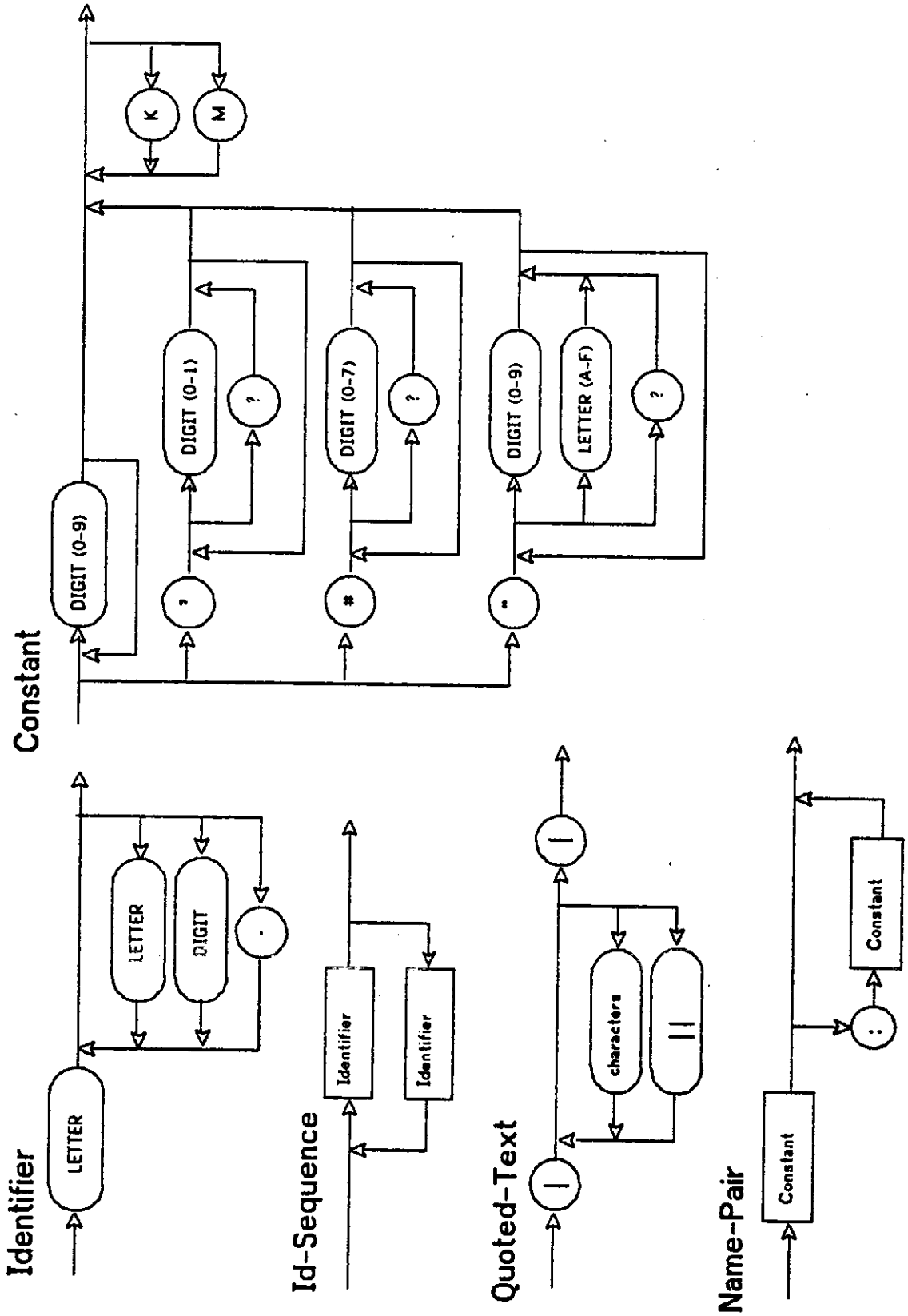


Figure 17-1: Syntax Chart - I

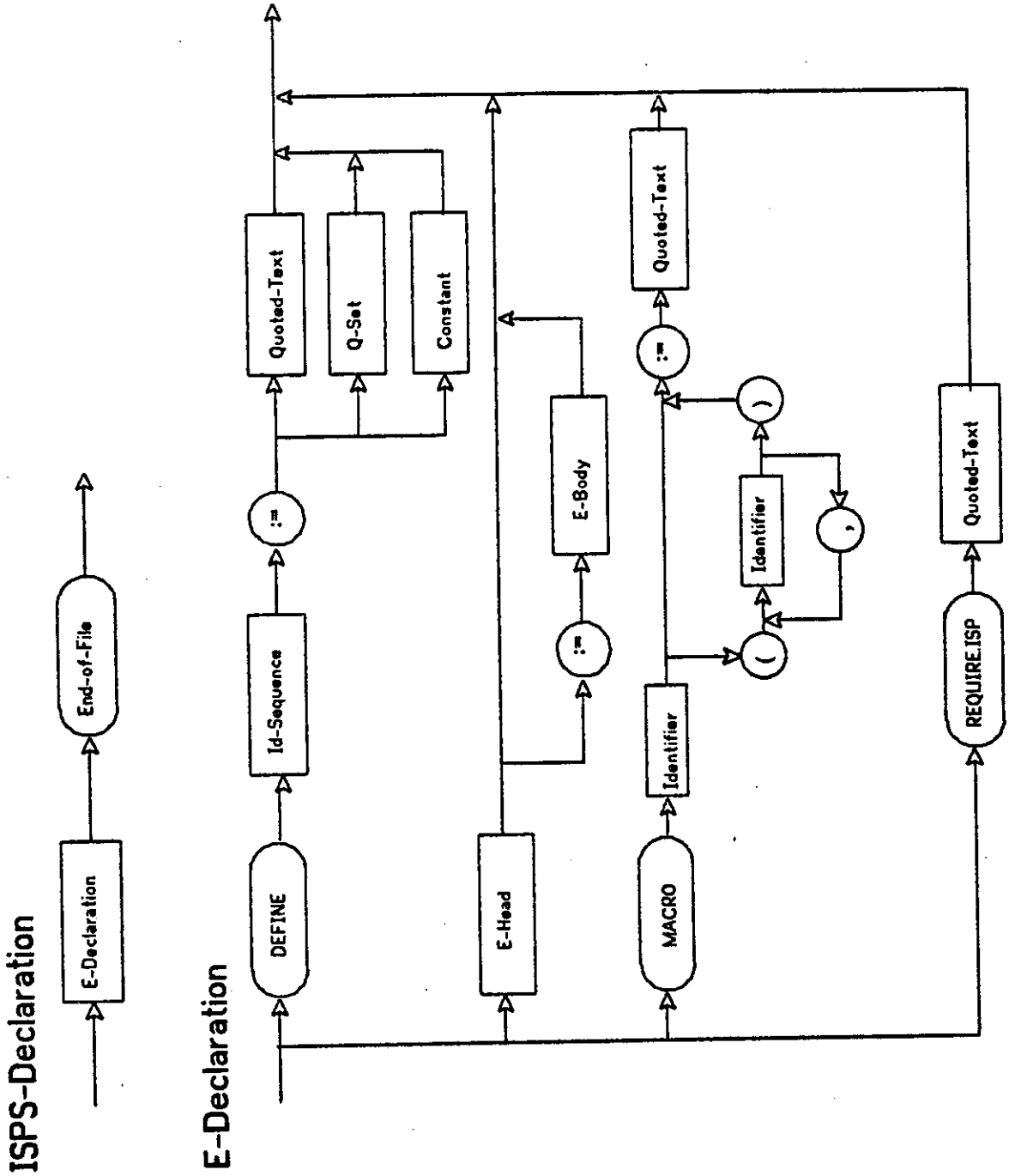


Figure 17-2: Syntax Chart - II

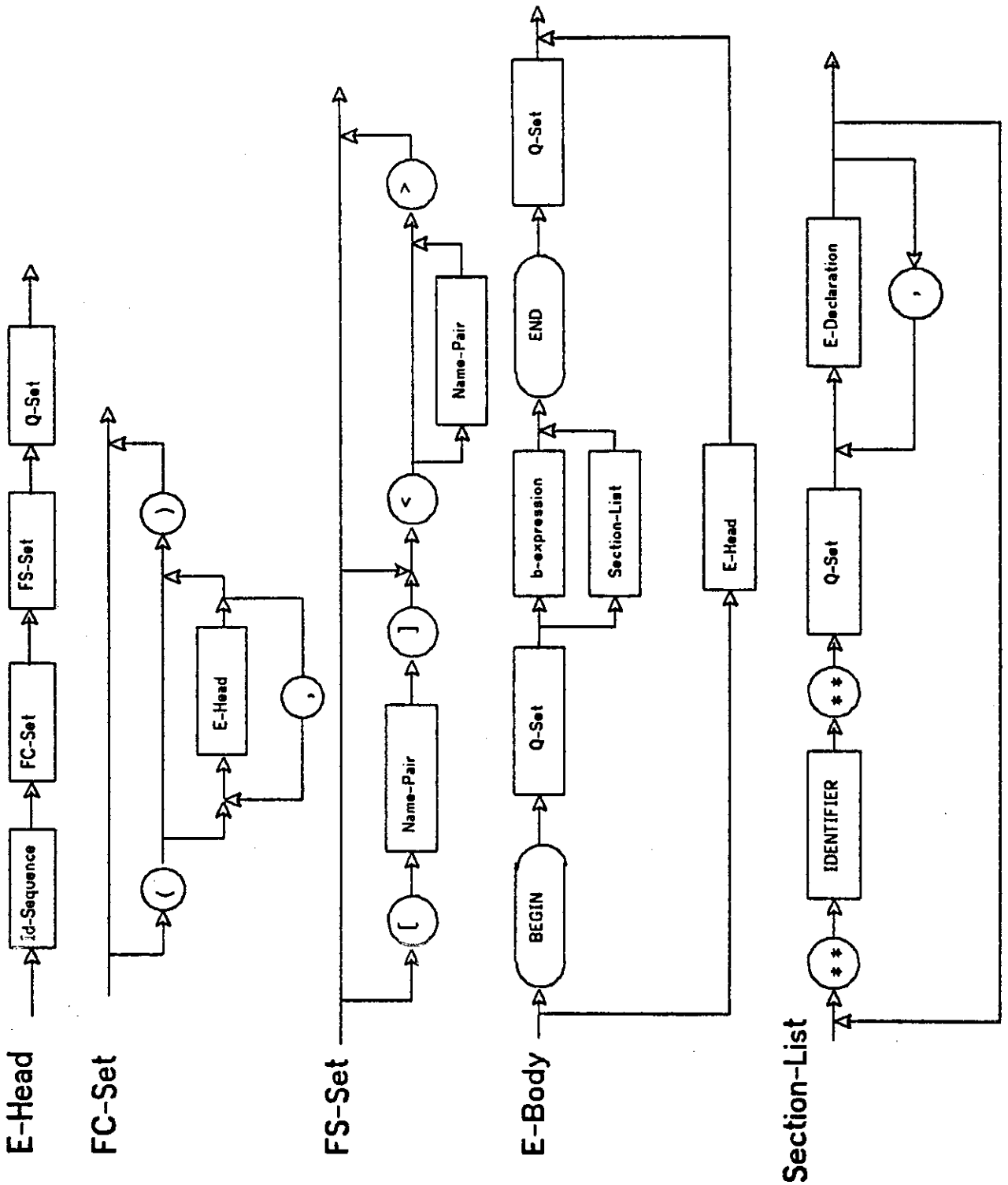


Figure 17-3: Syntax Chart - III

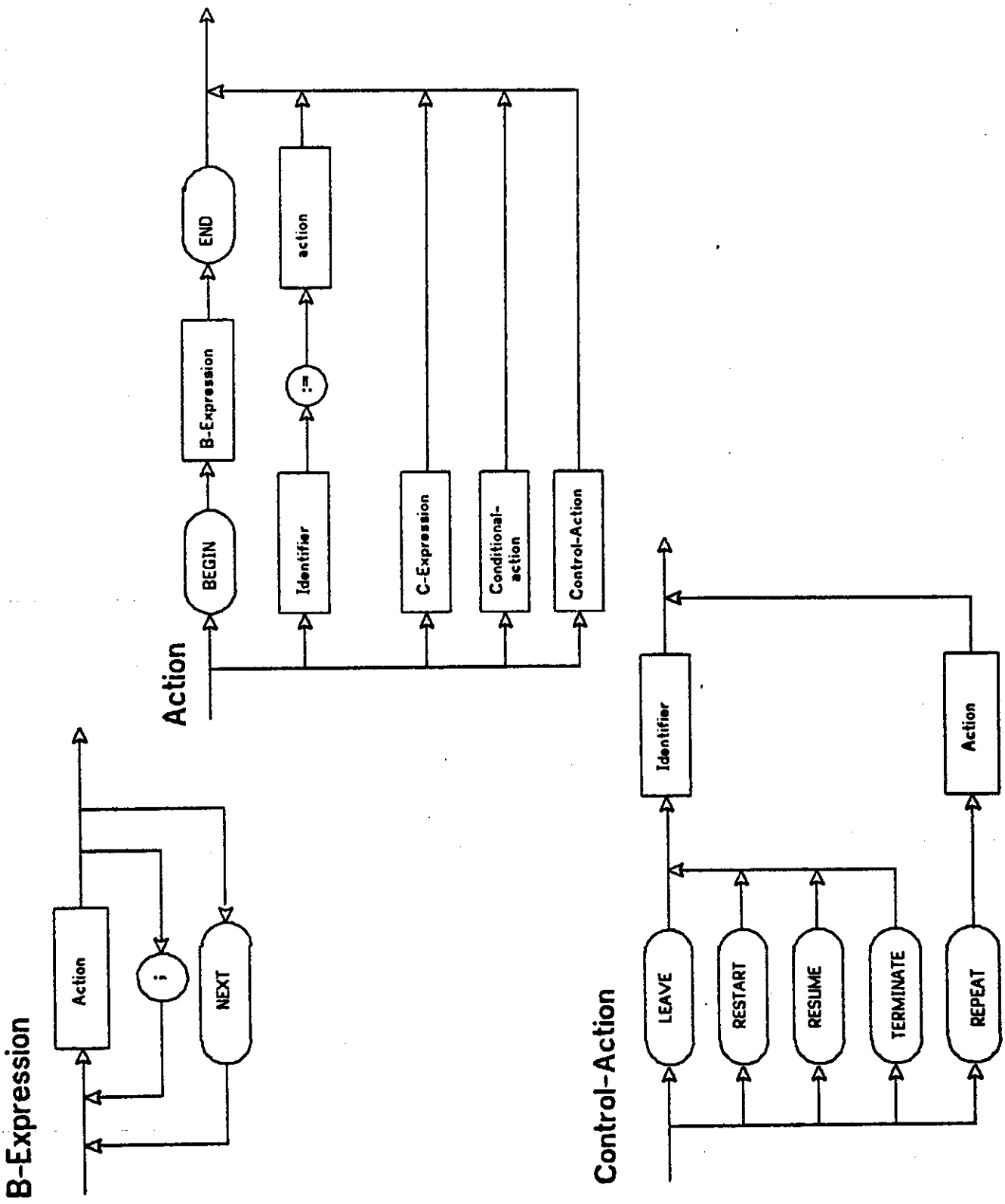


Figure 17-4: Syntax Chart - IV

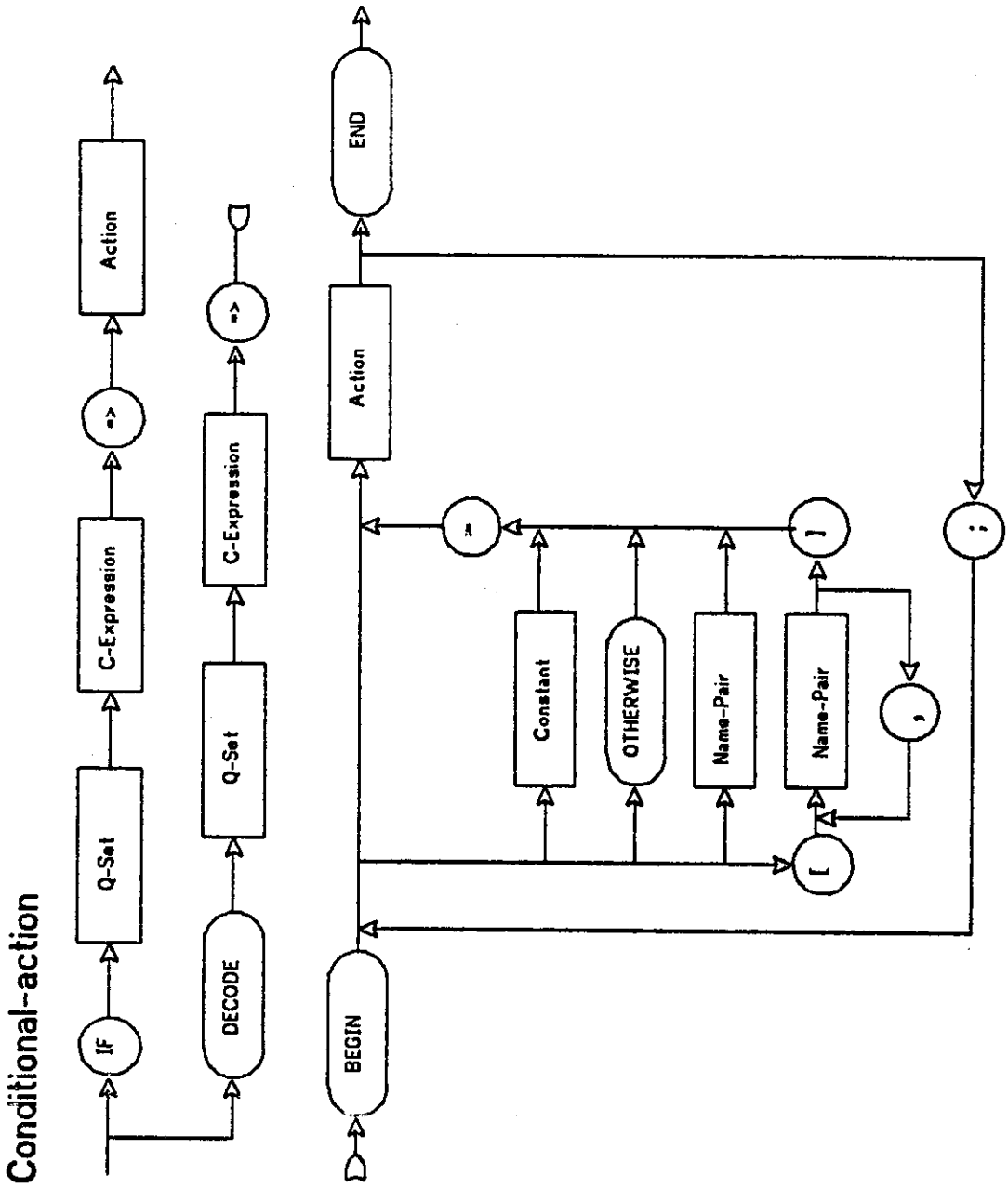


Figure 17-5: Syntax Chart - V

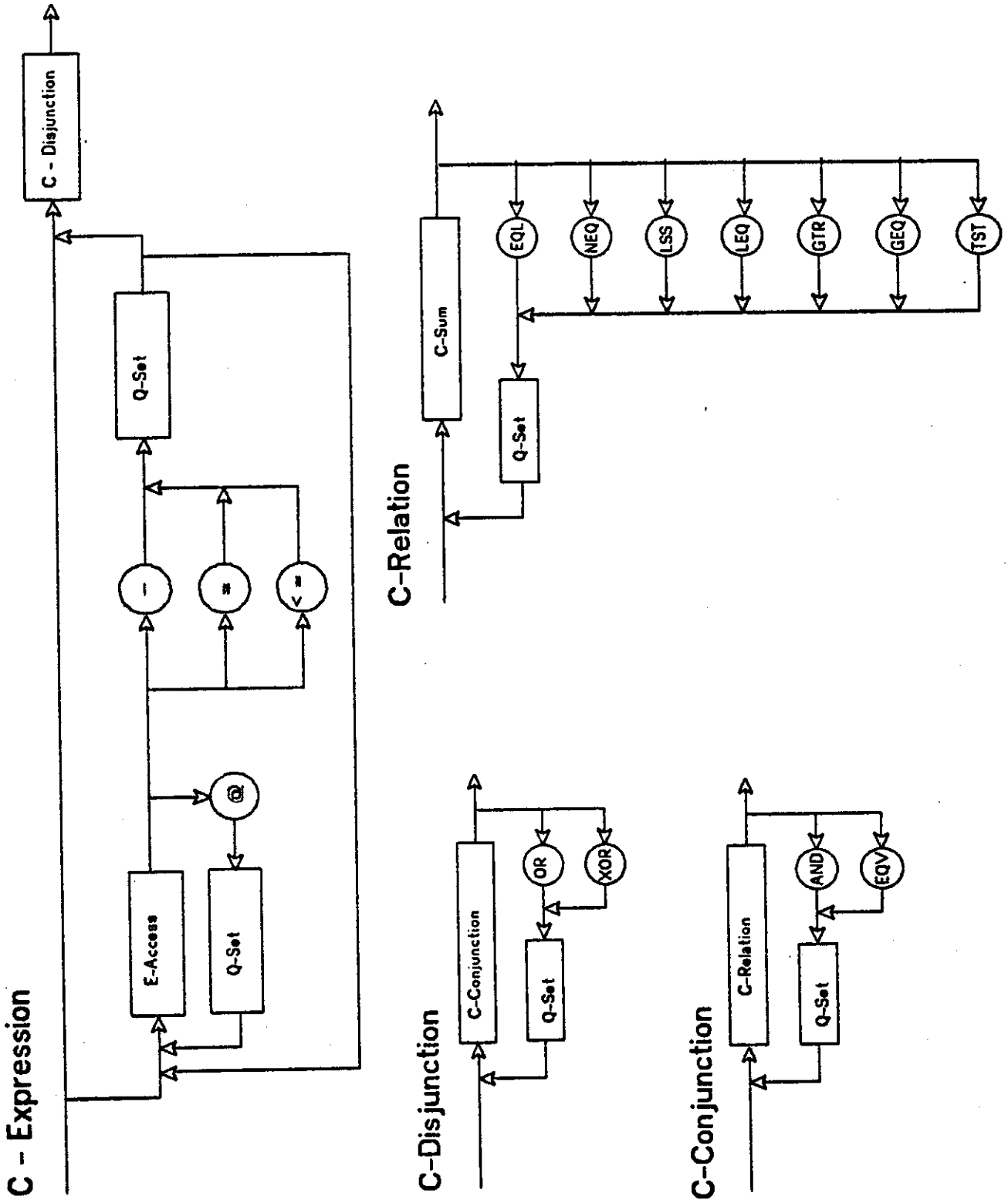


Figure 17-6: Syntax Chart - VI

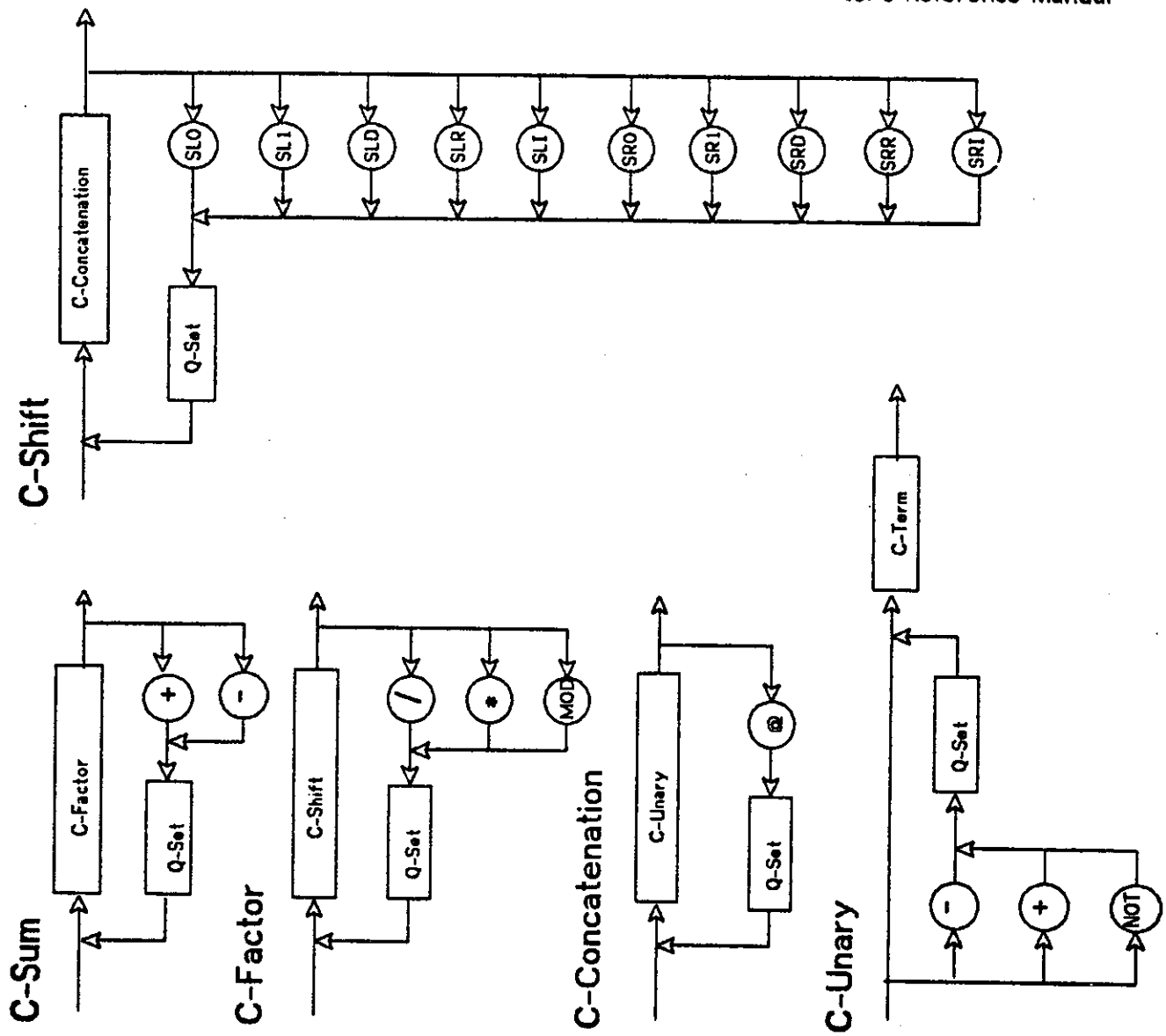


Figure 17-7: Syntax Chart - VII

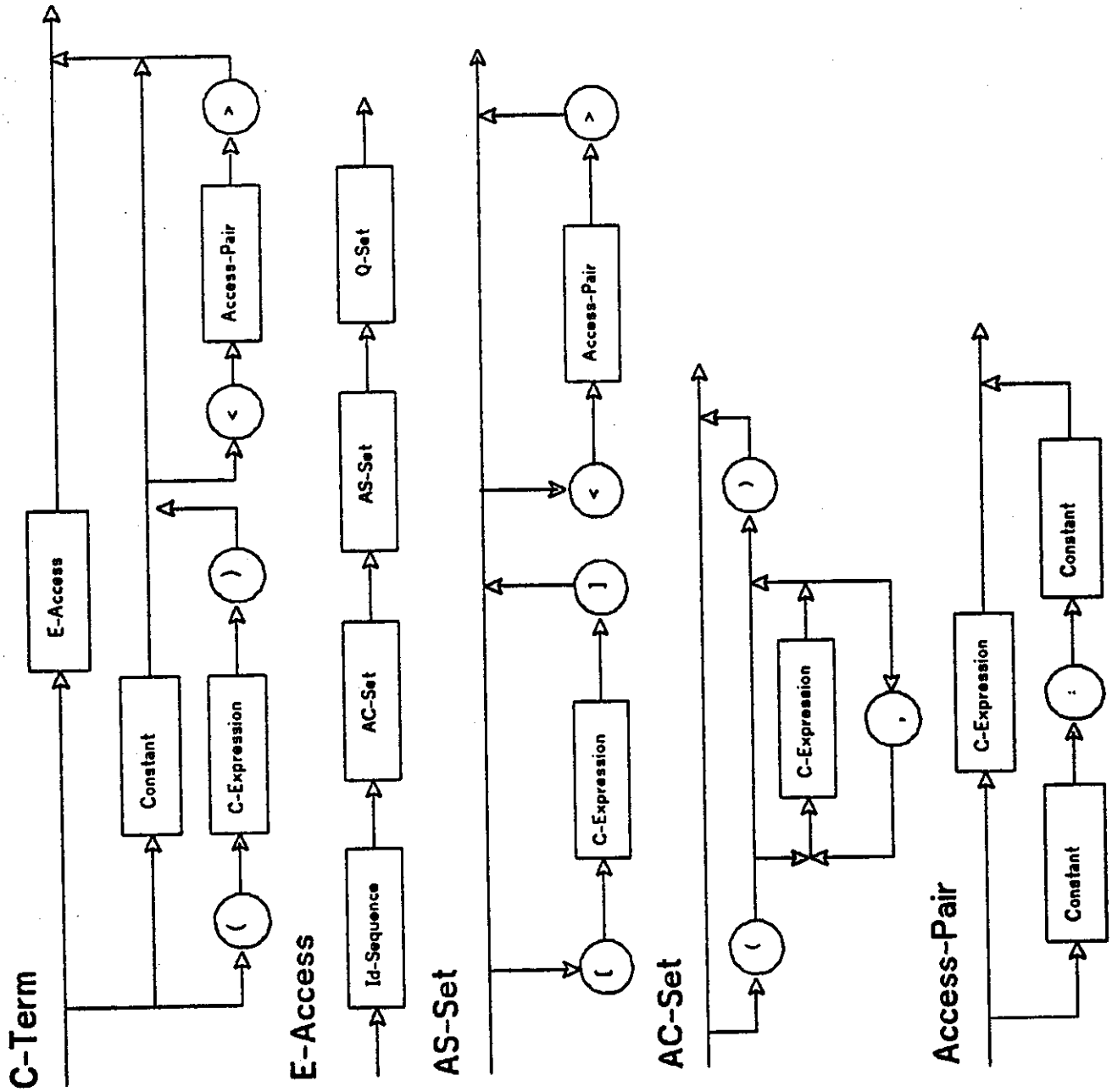


Figure 17-8: Syntax Chart - VIII

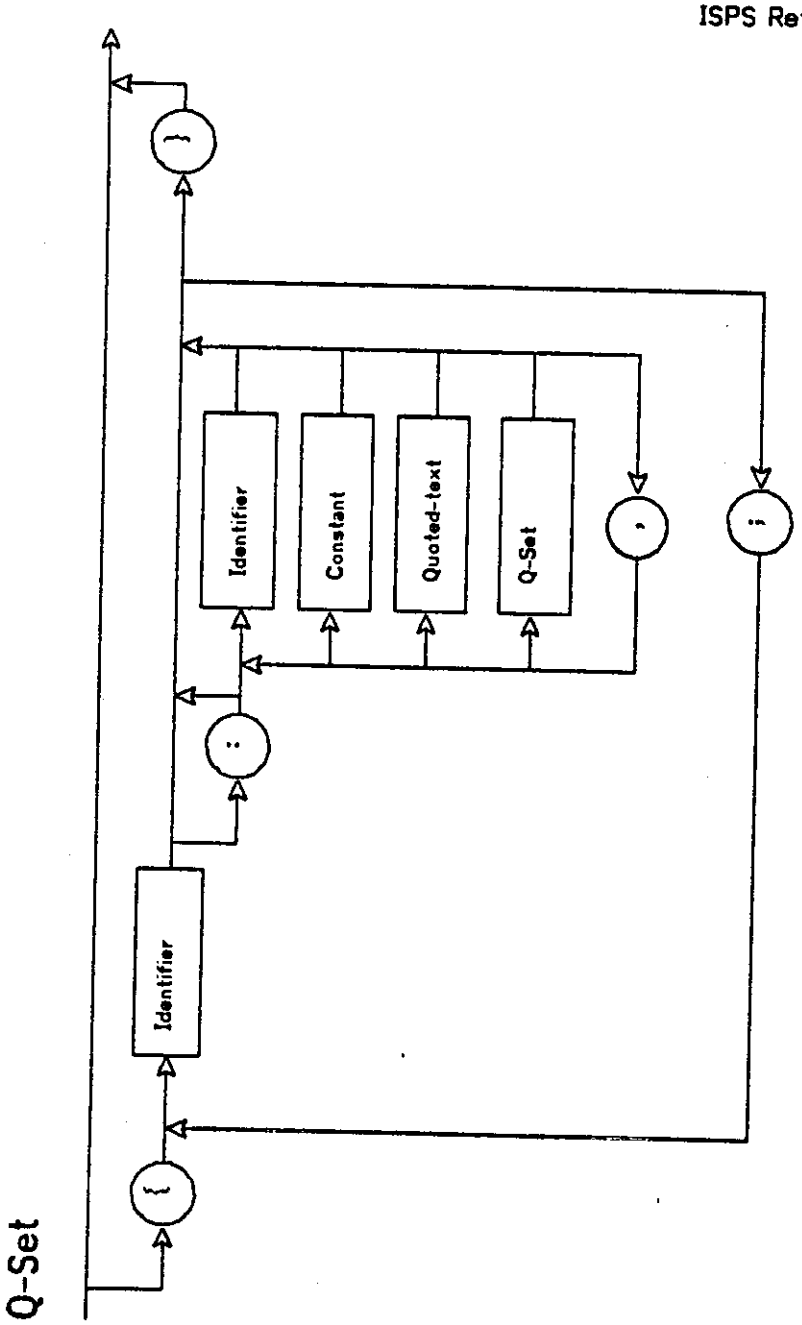


Figure 17-9: Syntax Chart - IX

Index