MIT/LCS/TR-218

VAL--A VALUE-ORIENTED ALGORITHMIC LANGUAGE: PRELIMINARY REFERENCE MANUAL

William B. Ackerman

Jack B. Dennis

June 13, 1979

This blank page was inserted to preserve pagination.

VAL -- A Value-Oriented Algorithmic Language :

Preliminary Reference Manual

by

William B. Ackerman

Jack B. Dennis

June 13, 1979

The language design reported herein was supported by the Lawrence Livermore Laboratory of the University of California under contract no. 8545403, and is based on work funded in part by the National Science Foundation under research grant DCR75-04060 and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract no. N00014-75-C-0661.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

VAL -- A Value-Oriented Algorithmic Language

1. INTRODUCTION

The programming language VAL (*Value-Oriented Algorithmic Language*) is designed for expressing algorithms for execution on computers capable of highly concurrent operation. More specifically, the application area to be supported is numerical computation which strains the limits of high performance machines, and the primary targets for translation of VAL programs are data driven machines of the form under development by the Computation Structures Group of the MIT Laboratory for Computer Science for high performance numerical computation.

Nevertheless, it has been our intention that the language not have idiosyncrasies reflecting the particular nature of the application area or target machine. It should be reasonable for VAL to evolve into a general purpose language appropriate for writing programs to run on future general purpose data flow computers.

In the design of VAL we have given careful consideration to the recently developed body of knowledge about program structures and language characteristics which support program verification. We have found a natural consistency between language design for support of concurrency and language design for correctness and verifiability. This has made it possible, in the design of VAL, to adhere to program structures and language characteristics that have been found desirable for ease of understanding and verification, and ease of building a program by combining separately specified modules.

We have undertaken the design of a new language because existing languages for numerical computation have a serious deficiency: they reflect the storage structure of the von Neumann concept of computer organization in that each language has some method of effecting a change in state of the memory which cannot be modeled as a local effect. Fortran, still the most popular language for large scale numerical work, is particularly blatant in this respect since it was conceived as a high level notation for programs to be run on a machine of classical design (the IBM 704).

Key words: programming languages, applicative programming, modularity

The difficulty with languages that allow specification of global state changes is that programs may be written which are very difficult or impossible to analyze for parts that may be executed concurrently. It is impossible in general to trace the flow of data with less than a complete analysis of the entire program. Only with such analysis is it possible to find and eliminate inessential constraints on the sequencing of program parts.

In contrast, the language VAL is entirely free of side effects: each module or well formed portion of a VAL program corresponds to a mathematical function and the entire effect of putting two parts together is to compose the corresponding functions. Such a language is *functional* or *applicative*. Although designs for applicative languages have been discussed many times in the literature, there have been few attempts to construct a complete and practical definition. This is due to the difficulty of incorporating file updates and input/output operations within the applicative framework, and the question of efficiency of implementation. The efficiency issue is countered in VAL by our goal of highly parallel execution, which is supported by applicative languages, and our aim to develop computer architectures specifically for efficient execution of programs expressed in functional languages.

The file update and input/output issues will be addressed in future versions of VAL in which streams of values will be introduced as a principal means for communicating between program modules. Modules that produce streams as output or accept streams as input can be used for input/output processes. Further, the implementation of transactions on a data base may be viewed as the processing of a stream of commands by a data base "secretary" or "guardian" module that holds the data base as internal data. If it is desired to realize more concurrency in processing transactions, the data base may be divided into parts, each with its own secretary module.

In developing the structure of VAL, it was natural for us to start from a language design which is of high quality, is well documented, and is close in spirit to our goals. Such a language is CLU [1, 2], developed at MIT by the Programming Methodology Group under Professor Barbara Liskov. In particular, CLU is designed for complete compile time type checking, and it has a set of well thought-out control structures and basic data types consonant with modern principles of structured programming.

While we have adopted many of the fundamental ideas of CLU, VAL differs radically from CLU in that the latter, like many new languages, is object oriented instead of value oriented. In keeping with this difference, the syntax and general structure of VAL are designed to reflect the functional character of the language and our desire to support highly concurrent program execution.

1.1 Acknowledgements

Current work on the development of VAL is funded by a grant from the Lawrence Livermore Laboratory of the University of California (LLL). We are thankful to Gus Dorough, George Michael and Lansing Sloan of LLL for their enthusiasm and support.

Several people have worked with us during the period of design of VAL, and have made major contributions to the language and this report: they are James McGraw and Charles Wetherell of LLL, and Dean Brock and Ken Weng of the MIT Computation Structures Group. James McGraw also produced the Syntax charts appearing at the end of this report. Others have influenced the development of VAL by suggesting features or requirements, and through their criticism of our documentation. These are Chris Hendrickson and Tim Rudy of LLL, and Andy Boughton, Randal Bryant, Clement Leung, Lynn Montz, and David Hirschman of the Computation Structures Group.

The ideas in the language grew out of our gradual self-education about data driven computation beginning around 1967. The students and staff of the Computation Structures Group who have contributed ideas include Earl van Horn, Peter Denning, Fred Luconi, Suhas Patil, Jorge Rodriguez, Chander Ramchandani, John Fosseen, Prakash Hebalkar, Jeffrey Gertz, Austin Henderson, Steve Zilles, Craig Schaffert, Eliot Moss, James Rumbaugh, David Misunas, David Isaman, Paul Kosinski, David Ellis, Sheldon Borkin, and Glen Miranker.

We thank the National Science Foundation for their continuing support, and acknowledge with appreciation the long period of support provided by the Advanced Research Projects Agency of the Department of Defense.

From outside MIT we have enjoyed support and inspiration especially from Joe Stoy of Oxford University, Gilles Kahn at IRIA, Paris, and Irvin Arvind of the University of California at Irvine.

In addition, we thank Barbara Liskov and her students for providing in CLU a convenient high quality starting point for our work on VAL.

1.2 References

[1] Liskov, B. H., et. al., "CLU Reference Manual", Computation Structures Group (Memo 161), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1978.

[2] Liskov, B. H., et. al., "Abstraction Mechanisms in CLU", *Communications of the ACM 20*, 8(August 1977), 564-576.

2. LANGUAGE SUMMARY

A program in VAL is a collection of separately translated parts called *modules*. Each module contains the definition of one external function. This function is accessible to all other modules of the VAL program by use of its name. A module may also contain the definitions of internal functions. These internal functions are used only within the module, and are not accessible to other modules.

The VAL language is applicative, that is, value-oriented. In contrast to many other languages, there are no "objects" thought of as residing in memory and being updated as the computation progresses. Even arrays and records are treated in VAL as mathematical values.

A function computes one or more data values as a function of one or more argument values. Except for invocations of other functions, a function invocation has access only to its arguments; there are no side effects. Further, a function retains no state information from one invocation to another; each function invocation is strictly independent. Hence values returned by a function depend only on the argument values presented to it -- a VAL function implements a true function in the mathematical sense.

The data types of VAL include the basic scalar types: boolean, integer, real, and character. Data structure values are either record values or array values. Records have a fixed format in which each field has a specified type. An array type has an integer index set and its components are of arbitrary but uniform type. Data structures of arbitrary depth may be specified using nested array and record types. Union types may be formed in which tags allow discrimination among a specified set of constituent types.

Each data type has its associated set of operations and predicates. Array and record types are treated as mathematical sets of values -- just as the boolean, integer, real, and character types. The operations for arrays and records are chosen to support identification of concurrency for execution on a highly parallel processor.

Exceptions are handled in VAL through special error elements in each data type. The element **undef** signals that one or more operand values are not in the specified domain of an operation. The element **miss_elt** signals that an array component is absent. Other error elements are provided in the numeric types to indicate arithmetic exceptions.

The design of VAL permits type checking to be performed by the translator. The type of each argument or result value of a function is specified in the function definition's header. Each value name used in the body of a function must have its data type specified. The operations of VAL are designed so that the types of the results can be determined if the types of the operands are known. Since the types of all atomic expressions are manifest, the types of all expressions can be determined.

Since VAL is a side-effect free language, subexpressions may be evaluated in any order without effect on computed results. Thus the control structures of VAL use a syntactic form -- an expression -- evaluation of which yields a tuple of values. Language constructs are provided for conditional expressions (**if/then/else**), and for iteration expressions (**for/iter**), the latter being a scheme for representing iterations as tail recursions. In addition, expression structures are provided for distributed computation of the components of a new array or of values to be combined by an operator. A **forall/construct** expression is used to compute the component values of a new array simultaneously. A **forall/eval** expression combines simultaneously computed values by an associative operation such as addition, multiplication, or maximum.

2.1 Notation

In the BNF presentation of the syntax, large curly braces $\{\ldots\}$ indicate zero or more repetitions of the material within. Large brackets $[\ldots]$ indicate that the material within may appear zero times or once.

3. PROGRAM FORMAT

Programs are written using the ASCII character set. No "control" characters other than tab and newline are used, except in character constants. The program elements are operation and punctuation symbols, real and integer numbers, character strings, reserved words, and names.

The operation and punctuation symbols are the following:

+	-	*	/	|	&
||	<	>	<=	>=	=
~=	:=	:	.	;	,
(	)	[	]	'	"

An integer number is a sequence of digits without a decimal point. A real number is a sequence of digits with either a decimal point or an exponent field. An exponent field is the letter "E" or "e", an optional sign, and one or more digits.

A character constant is a single character enclosed in single quotes. A character string constant is a string of zero or more characters enclosed in double quotes. Within each of these, tabulate, space, newline, percent, and all control characters represent themselves. A double quote may be placed in a string by using two double quote characters.

A reserved word is a word that always has a special meaning. Reserved words may never be used in any context for other than their special meaning. Reserved words in program examples and in the syntax are printed in **boldface** in this report.

The reserved words are:

abs	**do**	**if**	**over**
and	**else**	**in**	**plus**
arith_error	**elseif**	**integer**	**pos_over**
array	**endall**	**is**	**pos_under**
array_addh	**endfor**	**iter**	**real**
array_addl	**endfun**	**let**	**record**

array_adjust	endif	make	replace
array_empty	enditer	max	returns
array_fill	endlet	min	tag
array_join	endtag	miss_elt	tagcase
array_limh	error	mod	then
array_liml	eval	neg_over	times
array_remh	exp	neg_under	true
array_reml	external	nil	type
array_setl	false	null	undef
array_size	for	oneof	under
boolean	forall	or	unknown
character	function	otherwise	zero_divide
construct			

A name is a sequence of letters, digits, and underscores, of which the first character must be a letter. A name may not be the same as a reserved word. A name may be used as a value name, a function name, a defined type name, a record field name, or a oneof tag name. These uses all have their own mechanisms for interpretation, and hence a name may be used without conflict for several of these purposes. For example, a record field name occurs only in a record type specification or record operation, and hence will never be confused with a value name.

Upper and lower case letters in names and reserved words are not distinguished, but all uses of a name or reserved word must have consistent capitalization. Names may be of any reasonable length.

The separating characters space, tabulate, and newline are equivalent (except in delimiting comments), and may appear anywhere except within a program element. Hence they may not appear within a number or between the characters of a two character operation symbol such as >=. A separating character is required only between adjacent constants, names, or reserved words. For example, separating characters are required to distinguish the program construct "if p then 3 else 4 endif" from the name "ifpthen3else4endif". Separating characters not required next to operation or punctuation symbols.

A comment begins with a percent sign and continues to the end of the line. A comment is equivalent to a space, and hence may be placed anywhere except within an program element.

Examples of names and constants:

```
ABC3_Q
34
.3141593E1
2.718282
5772157E-7
'%'
''
"abc""def"
```

4. VALUES AND TYPES

The inputs and outputs of VAL expressions and functions are values. The entire collection of values that may be presented to or produced by VAL programs is the value domain of VAL. The value domain is subdivided into distinct disjoint subdomains that are the data types of VAL. There are *basic types* which include the familiar scalar values of computation; *structured types* in the form of arrays and records as defined by the language user in terms of simpler data types; and discriminated union types.

4.1 Type Specifications

A type specification in VAL is a syntactic construct that specifies a data type.

Syntax:

```
type-spec ::= basic-type-spec
            | compound-type-spec
            | type-name
basic-type-spec ::= null | boolean | integer | real | character
compound-type-spec ::= array [type-spec]
            | record [ field-spec { ; field-spec } ]
            | oneof [ tag-spec { ; tag-spec } ]
field-spec ::= field-name { , field-name } : type-spec
tag-spec ::= tag-name { , tag-name } [ : type-spec ]
field-name ::= name
tag-name ::= name
type-name ::= name
```

For a basic type, the specification is simply the name of the type. For a compound type, the specification consists of a *type constructor* giving the name of the compound type followed by the necessary additional information within brackets.

The array type constructor gives the type of the elements of the array.

Examples:

```
array [ integer ]
array [ array [ real ]]
```

The record type constructor gives the field names and the type associated with each field. The field names used within any record specification must be distinct. Where several field names are listed with one type, the fields are all of that type.

Examples:

```
record [ I, J : integer ; TEMP : real ]
record [ I : record [ X : array [ boolean ] ; Y : character ] ; TEMP : real ]
```

A name may be used as a field name and as any other name (but not a reserved word) without conflict, since it is interpreted as a field name only in the record constructor and in record operations. The same field name may be used in several record types without conflict.

The **oneof** (union) type constructor gives the tags and the type associated with each tag. The tag names must be distinct. Where several tag names are listed with one type, the tags all indicate that type. If the colon and following type specification are omitted, the **null** type is assumed.

Examples:

```
oneof [ UP, DOWN, LEFT, RIGHT ]
oneof [ FIX : integer ; FLO : real ]
oneof [ THIS : array [ integer ] ; THAT, THE_OTHER : record [ C : real ; D : boolean ]]
```

As in the case of field names, a tag name may coincide with any other name without conflict, and the same tag name may be used in several union types without conflict.

Any type name used as a type specification must be defined by a type definition (see Section 4.6).

4.2 Value Domains

Each data type is a domain of values as described below. As will be seen, each data type includes proper elements, and error elements which occur as the result of an expression when computation of a proper value of the type is impossible. Each data type is further characterized by the set of operations that may be used to create and transform values of the type. The operations for each data type of VAL are defined in Section 5, as are conversion operations that convert values of one type into values of another.

4.3 Error Values

The error elements are included to support the unusual treatment of exceptions adopted in VAL as discussed in Sections 5 and 7. The full name of an error value consists of an error name followed by the type specification enclosed in brackets, for example **zero_divide[real]**. This is because every value, including all error values, must have a unique type, so, for example, **zero_divide[real]** is a different value from **zero_divide[integer]**.

Two error values are members of every data type: the element **undef**[type] results when operand values are not in the domain of an operator, for example, if the index of an array access operation is outside the range of the array; the element **miss_elt**[type] results if the index of an array access operation is within the array range, but no data value exists at that index.

4.4 Basic Types

The Null Type

proper elements: **nil**

error elements: **undef[null]**, **miss_elt[null]**

The null type occurs in a distinguished union (**oneof**) type where in one or more alternatives no data value is required.

The Boolean Type

proper elements: **true, false**

error elements: **undef[boolean], miss_elt[boolean]**

The Integer Type

proper elements: The integers between some limits which are implementation dependent.

error elements: **undef[integer], miss_elt[integer], pos_over[integer], neg_over[integer], unknown[integer], zero_divide[integer]**

The elements **pos_over[integer]** and **neg_over[integer]** indicate that the integer value is too large (positive or negative) to be represented in the implementation. The element **unknown[integer]** indicates the result of a computation that has exceeded the capacity of the implementation, but whose true value is not known to be out of range. The element **zero_divide[integer]** indicates the result of a division or modulus operation with zero divisor.

The Real Type

proper elements: Floating point representations of real numbers including zero, with some exponent range which is implementation dependent.

error elements: **undef[real], miss_elt[real], pos_over[real], neg_over[real], pos_under[real], neg_under[real], unknown[real], zero_divide[real]**

The elements **pos_over[real]** and **neg_over[real]** indicate that the real value is larger (positive or negative) than is representable in the floating point method of the implementation. The elements **pos_under[real]** and **neg_under[real]** represent non-zero values too small in magnitude to be representable in the floating point method of the implementation. The element **unknown[real]** indicates the result of a computation that has exceeded the exponent range of the implementation, but whose true value is not known to be out of range. The element

zero_divide[real] indicates the result of an attempted division by zero.

The Character Type

proper elements: The 128 characters of the ASCII character set.

error elements: **undef[character]**, **miss_alt[character]**

4.5 Compound Types

Array Types

For each data type defined by some VAL type specification T, an *array type* may be defined by the type specification **array[T]**.

proper elements: A proper array value in **array** [T] consists of two components:

(1) A *range* (LO, HI) where LO and HI are integers and LO $\leq$ HI + 1. These are inclusive bounds on the defined elements. If LO = HI + 1 the array has no elements.

(2) A sequence of HI - LO + 1 elements of type T.

error elements: Every array type **array[T]** includes the elements **undef[array[T]]** and **miss_alt[array[T]]**.

Record Types

If $t_1, \ldots, t_k$ are VAL type specifications and $n_1, \ldots, n_k$ are distinct names, then **record** [$n_1 : t_1 ; \ldots ; n_k : t_k$] specifies a record type.

proper elements: Each proper value of the record type is a set of k pairs $\{(n_1, v_1), \ldots, (n_k, v_k)\}$ where each v_i is an element of t_i.

error elements: **undef[T]**, **miss_alt[T]**, where T is the record type

Union Types

Each element of a union type is an element of one of several constituent types, accompanied by a tag which indicates the constituent type from which the element was taken. If $t_1, \ldots, t_k$ are type specifications, and $n_1, \ldots, n_k$ are distinct names, then **oneof** [n_1 : t_1 ; . . . ; n_k : t_k] specifies a union type.

proper elements: Each proper element of the union type is a pair (n_i, v_i) where $1 \leq i \leq k$ and v_i is an element of t_i.

error elements: **undef**[T], **miss_elt**[T], where T is the union type

4.6 Type Definitions

Syntax:

type-def ::= **type** type-name = type-spec

type-name ::= name

A function definition may contain a number of type definitions which specify programmer-named types used in the function. Each type definition specifies that a type name denotes the type represented by the given type specification. The type specification part of a type definition may contain type names defined in the same or other definitions. Recursion and mutual recursion are permitted in type definitions. Such type definitions may be used to construct data types composed of array or record structures of unlimited depth.

Example:

type STACK = **oneof** [empty : **null** ; element : **record** [value : **real** ; rest : STACK]] ;

The name of a defined type may be used anywhere that a type specification is permitted, e.g. as the type parameter for constants such as **miss_elt**[type-spec].

A name may be used as a type name and as any other kind of name without conflict, since it is interpreted as a type name only in well defined contexts.

4.7 Equivalence of Type Specifications

Type checking is performed by the VAL translator by testing that the type of each expression or subexpression matches the type required by the context in which it appears. The type of an expression or subexpression is determined by its composition from operators and elementary terms as described in Sections 5 and 6. This must match the type required by its context: an argument to a function must match the argument type indicated in the function's definition, and an expression on the right hand side of a definition (see Section 7.2) must match the declared type of the name on the left hand side.

The necessary test is to determine if two type specifications are *equivalent*, that is, if they denote the same type. Two basic type specifications are equivalent if they are the same. Two **array** specifications are equivalent if their element types are equivalent. Two **record** or **oneof** type specifications are equivalent if their correspondingly named component types or constituent types are equivalent; the order in which they are listed is not significant. A defined type name is equivalent to the type appearing on the right hand side of its definition.

A compound type specification can be visualized as a tree whose nodes are labeled **array**, **record**, or **oneof**, whose arcs from **record** or **oneof** nodes are labeled with field or tag names, and whose leaves are basic types. Equivalence can be formulated in terms of this characterization: Two type specifications are equivalent if their trees are identical, disregarding the order of arcs. If a type specification uses recursion, this tree is infinite; two such specifications are equivalent if these infinite trees are identical.

Examples -- assume the following type definitions:

```
type NUM = real ;
type STACK = oneof [ empty : null ; element : ITEM ] ;
type ITEM = record [ value : real ; rest : STACK ] ;
```

Then the following pairs of type specifications are equivalent:

real	(A defined type is exactly equivalent
NUM	to the type that it is defined to be.)
record [a : **real** ; b : **integer**]	(order of fields is not significant)
record [b : **integer** ; a : **real**]	
oneof [empty : **null** ; element : **record** [value : **real** ; rest : STACK]] ;	
STACK	(The (infinite) trees implied by these type specifications are equivalent.)

5. OPERATIONS

In this section we specify the sets of operations applicable to each data type of VAL. In the examples of notation, P and Q stand for boolean values, J and K for integers, X and Y for reals, C and D for characters, A and B for arrays, R for records, U for union (oneof) values, and V for values of arbitrary type.

5.1 Error tests

A number of tests are provided for error elements. The following three are defined for all types:

operation	notation	functionality
test for **undef**	**is undef**(V)	any → bool
test for **miss_elt**	**is miss_elt**(V)	any → bool
test for all errors	**is error**(V)	any → bool

The test **is error** is satisfied by all error values for the type to which it is applied: **undef**, **miss_elt**, and any other errors such as **zero_divide** that exist for that type. Additional error tests, such as **is over**, are defined below for certain types.

All error test operations always return **true** or **false**, never an error value. They must be used for testing for errors in preference to the equality operator (e.g. "X = **undef**[**real**]"), since the latter returns **undef**[**boolean**] when X is an error value.

5.2 Null operations

The **null** type is used to provide a case in a union type for which the value is irrelevant. There are no operations for this type except the error tests **is undef**, **is miss_elt**, and **is error**.

5.3 Boolean operations

The boolean operations are the following:

operation	notation	functionality
and	P & Q	bool, bool → bool
or	P | Q	bool, bool → bool
not	~ P	bool → bool
equal	P = Q	bool, bool → bool
not equal	P ~= Q	bool, bool → bool
test for **undef**	**is undef**(P)	bool → bool
test for **miss_elt**	**is miss_elt**(P)	bool → bool
test for **undef** or **miss_elt**	**is error**(P)	bool → bool

If an error value is an operand to a boolean operation other than an error test, the result is **undef[boolean]**.

5.4 Integer operations

The integer operations are the following:

operation	notation	functionality
addition	J + K	int, int → int
subtraction	J - K	int, int → int
multiplication	J * K	int, int → int
division	J / K	int, int → int
modulus	**mod**(J, K)	int, int → int
exponentiation	**exp**(J, K)	int, int → int
negation	- J	int → int

magnitude	abs(J)	int → int
maximum	max(J, K)	int, int → int
minimum	min(J, K)	int, int → int
equal	J = K	int, int → bool
not equal	J ~= K	int, int → bool
greater, less	J > K, J < K	int, int → bool
greater/equal, less/equal	J >= K, J <= K	int, int → bool
test for pos_over	is pos_over(J)	int → bool
test for neg_over	is neg_over(J)	int → bool
test for unknown	is unknown(J)	int → bool
test for zero_divide	is zero_divide(J)	int → bool
test for pos_over or neg_over	is over(J)	int → bool
test for pos_over, neg_over, unknown, or zero_divide	is arith_error(J)	int → bool
test for undef	is undef(J)	int → bool
test for miss_elt	is miss_elt(J)	int → bool
test for undef, miss_elt, pos_over, neg_over, unknown, or zero_divide	is error(J)	int → bool

The error value zero_divide[integer] may result from the division or modulus operations. The error values pos_over[integer] or neg_over[integer] may result from the arithmetic operations if the result exceeds the range of numbers representable on the target computer.

If the error value undef[integer], miss_elt[integer], or zero_divide[integer] is an operand to any integer operation other than an error test, the result is undef of the appropriate type.

The integer operators have the following special behavior with respect to the error values **pos_over**, **neg_over**, and **unknown**. These rules are of course symmetric with respect to exchange of the arguments to +, *, **max**, and **min**. These rules do *not* apply if any operand is **undef**, **miss_elt**, or **zero_divide**.

1a. **pos_over** + J = **pos_over** if J ≥ 0 or J = **pos_over**,
unknown otherwise

1b. **neg_over** + J = **neg_over** if J ≤ 0 or J = **neg_over**,
unknown otherwise

1c. **unknown** + J = **unknown**

2a. - **pos_over** = **neg_over**

2b. - **neg_over** = **pos_over**

2c. - **unknown** = **unknown**

3. J - K = J + (- K), so, for example, by rules 2a and 1b,
J - **pos_over** = **neg_over** if J ≤ 0 or J = **neg_over**,
unknown otherwise

4a. J * **pos_over** = **neg_over** if J ≤ -1 or J = **neg_over**,
pos_over if J ≥ 1 or J = **pos_over**,
0 if J = 0,
unknown otherwise

4b. J * **neg_over** = - (J * **pos_over**)

4c. J * **unknown** = 0 if J = 0,
unknown otherwise

5a. J < **pos_over** = **true** unless J = **pos_over** or **unknown**,
in which case the result is **undef**

5b. **neg_over** < J = **true** unless J = **neg_over** or **unknown**,
in which case the result is **undef**

The preceding two rules also yield **true** if the connective is <=, and **false** if the connective is > or

>=. They are also of course symmetric with respect to exchange of the arguments and reversal of the connective.

6a. abs(pos_over) = abs(neg_over) = pos_over

6b. abs(unknown) = unknown

7a. max(pos_over, J) = pos_over

7b. min(pos_over, J) = J

7c. max(neg_over, J) = J

7d. min(neg_over, J) = neg_over

7e. max(unknown, J) = unknown

7f. min(unknown, J) = unknown

Other than the above cases, if any operand to a integer operation other than an error test is an error value, the result is undef of the appropriate type.

5.5 Real operations

The real operations are the following:

operation	notation	functionality
addition	X + Y	real, real → real
subtraction	X - Y	real, real → real
multiplication	X * Y	real, real → real
division	X / Y	real, real → real
exponentiation	exp(X, Y)	real, real → real
exponentiation with integer	exp(X, J)	real, int → real
negation	- X	real → real
magnitude	abs(X)	real → real
maximum	max(X, Y)	real, real → real
minimum	min(X, Y)	real, real → real

equal	X = Y	real, real → bool
not equal	X ~= Y	real, real → bool
greater, less	X > Y, X < Y	real, real → bool
greater/equal, less/equal	X >= Y, X <= Y	real, real → bool

---

test for pos_over	is pos_over(X)	real → bool
test for neg_over	is neg_over(X)	real → bool
test for pos_under	is pos_under(X)	real → bool
test for neg_under	is neg_under(X)	real → bool
test for unknown	is unknown(X)	real → bool
test for zero_divide	is zero_divide(X)	real → bool
test for pos_over or neg_over	is over(X)	real → bool
test for pos_under or neg_under	is under(X)	real → bool
test for pos_over, neg_over, pos_under, neg_under, unknown, or zero_divide	is arith_error(X)	real → bool

---

test for undef	is undef(X)	real → bool
test for miss_elt	is miss_elt(X)	real → bool
test for undef, miss_elt, pos_over, neg_over, pos_under, neg_under, unknown, or zero_divide	is error(X)	real → bool

---

The error value zero_divide[real] may result from the division operation. The error values pos_over[real], neg_over[real], pos_under[real], or neg_under[real] may result from the arithmetic operations if the result exceeds the range of numbers representable on the target computer.

If the error value undef[real], miss_elt[real], or zero_divide[real] is an operand to any real operation other than an error test, the result is undef of the appropriate type.

The real operators have the following special behavior with respect to the error values pos_over, neg_over, and unknown. These rules are of course symmetric with respect to exchange of the arguments to +, *, max, and min. These rules do not apply if any operand is undef, miss_elt, or zero_divide.

1a. pos_over + X = pos_over if X ≥ 0.0 or X = pos_over or pos_under,
unknown otherwise

1b. neg_over + X = neg_over if X ≤ 0.0 or X = neg_over or neg_under,
unknown otherwise

1c. unknown + X = unknown

1d. pos_under + X = X if X ≠ 0.0 and is a proper value

1e. neg_under + X = X if X ≠ 0.0 and is a proper value

1f. pos_under + pos_under = pos_under

1g. neg_under + neg_under = neg_under

1h. pos_under + neg_under = unknown

1i. pos_under + 0.0 = pos_under

1j. neg_under + 0.0 = neg_under

2a. - pos_over = neg_over

2b. - neg_over = pos_over

2c. - pos_under = neg_under

2d. - neg_under = pos_under

2e. - unknown = unknown

3. X – Y = X + (- Y), so, for example, by rules 2a and 1b,
X - pos_over = neg_over if X ≤ 0.0 or X = neg_over or neg_under,
unknown otherwise

4a. X * pos_over = neg_over if X ≤ -1.0 or X = neg_over,
pos_over if X ≥ 1.0 or X = pos_over,
0.0 if X = 0.0,
unknown otherwise

4b. X * neg_over = - (X * pos_over)

4c. X * pos_under = neg_under if $-1.0 \leq X < 0.0$ or X = neg_under,
pos_under if $0.0 < X \leq 1.0$ or X = pos_under,
0.0 if X = 0.0,
unknown otherwise

4d. X * neg_under = - (X * pos_under)

4e. X * unknown = 0.0 if X = 0.0,
unknown otherwise

5a. X < pos_over = true unless X = pos_over or unknown,
in which case the result is undef

5b. neg_over < X = true unless X = neg_over or unknown,
in which case the result is undef

The preceding two rules also yield true if the connective is <=, and false if the connective is > or >=. They are also of course symmetric with respect to exchange of the arguments and reversal of the connective.

6a. abs(pos_over) = abs(neg_over) = pos_over

6b. abs(pos_under) = abs(neg_under) = pos_under

6c. abs(unknown) = unknown

7a. max(pos_over, X) = pos_over

7b. min(pos_over, X) = X

7c. max(neg_over, X) = X

7d. min(neg_over, X) = neg_over

7e. max(unknown, X) = unknown

7f. min(unknown, X) = unknown

Other than the above cases, if any operand to a real operation other than an error test is an error value, the result is undef of the appropriate type.

5.6 Character operations

The character operations are the following:

operation	notation	functionality
equal	C = D	char, char → bool
not equal	C ~= D	char, char → bool
test for undef	is undef(C)	char → bool
test for miss_elt	is miss_elt(C)	char → bool
test for undef or miss_elt	is error(C)	char → bool

If an error value is an operand to a character operation other than an error test, the result is undef[character].

5.7 Array operations

The operations for the array data type array[T] include creation of new arrays, selection, producing new array values by appending components to an array value, and combining arrays by concatenation. Recall that an array value consists of a range defined by a low index LO, a high index HI, and a sequence of HI-LO+1 elements of the given type, some of which may be miss_elt[T].

operation	notation	functionality
create	array_empty[T]	→ array[T]
create/fill	array_fill(J, K, V)	int, int, T → array[T]
select	A[J]	array[T], int → T
append	A[J : V]	array[T], int, T → array[T]
create by elements	[J : V]	int, T → array[T]
index of highest	array_limh(A)	array[T] → int

index of lowest	array_limi(A)	array[T] → int
number of elements	array_size(A)	array[T] → int
set bounds	array_adjust(A, J, K)	array[T], int, int → array[T]
extend high	array_addh(A, V)	array[T], T → array[T]
extend low	array_addl(A, V)	array[T], T → array[T]
remove high	array_remh(A)	array[T] → array[T]
remove low	array_reml(A)	array[T] → array[T]
set low limit	array_setl(A, J)	array[T], int → array[T]
concatenate	A ‖ B	array[T], array[T] → array[T]
merge defined elements	array_join(A, B)	array[T], array[T] → array[T]
test for undef	is undef(A)	array[T] → bool
test for miss_elt	is miss_elt(A)	array[T] → bool
test for undef or miss_elt	is error(A)	array[T] → bool

In general, the result of an array operation is the error element undef of the appropriate type if either an index operand is an error value or an array operand is undef or miss_elt. The remaining cases in which the result is an error are specified below for each operation.

Create array_empty[type-spec]

This is actually a constant. It is an array of the indicated type, whose low index is one, high index is zero, and which therefore contains no elements.

Create/fill array_fill(LO, HI, V)

This creates an array with the given range and all elements equal to the given value. If LO > HI+1, the result is undef[array[T]]. This operation yields a proper array even if V is an error value such as undef or pos_over.

Example:

array_fill(1, 10, 6)

is an array with 10 elements, all equal to 6.

Select A[J]

This operation yields the element of the array A at index J. If J is not within the range of the array, the result is undef[T]. Otherwise, the result is whatever value is in the array, which may be an error value such as miss_elt[T] or undef[T].

Append A [J : V]

This returns an array identical to A except that the element at index J has been replaced by value V. The range of A is expanded as needed to include index J, and any new elements in the expanded range are given the value miss_elt[T]. For example, if A has bounds 1 and 3, and J is 10, elements 4 through 9 will be miss_elt[T] in the result.

Create by elements [J : V]

This returns an array with low and high indices both J, and one element V at index J.

There are abbreviated notations for compositions of *select*, *append*, and *create by elements* operations to simplify construction of multiple element arrays and for operating on multi-dimensional arrays. See Section 6.4.

Index of highest, lowest array_limh(A), array_liml(A)

These functions return the high or low index of A, respectively.

Number of elements **array_size**(A)

This returns **array_limh**(A) - **array_liml**(A) + 1.

Set bounds **array_adjust**(A, J, K)

This returns an array with range (J, K), containing the same data as A where possible. If J is greater than **array_liml**(A) or K is less than **array_limh**(A), some elements of A will be absent in the result. If J is less than **array_liml**(A) or K is greater than **array_limh**(A), the newly created positions are set to **miss_elt**[T].

Extend high, low **array_addh**(A, V), **array_addl**(A, V)

These return the array A with its high index increased by one or its low index decreased by one, and the given value V as the new element.

Remove high, low **array_remh**(A), **array_reml**(A)

These return the array A with its high index decreased by one or its low index increased by one. An element of A is lost in the result. If the array A has size zero, the result is **undef**.

Set low limit **array_setl**(A, J)

This adds J - **array_liml**(A) to all element indices and to both components of the range, yielding an array similar to A but with the origin shifted. Its low index is J.

array_setl([2 : X, Y, Z], 5)

denotes the same value as

[5 : X, Y, Z]

where the abbreviated notation is defined in Section 6.4.

Concatenate A || B

This returns an array whose size is the sum of the sizes of A and B, formed by concatenating A and B. The low index of the result is the same as the low index of A, and the elements of A retain their original indices. The indices of B are shifted as necessary.

Merge defined elements **array_join**(A, B)

This merges the arrays by elements. The low index of the result is the minimum of **array_liml**(A) and **array_liml**(B), and the high index is the maximum of **array_limh**(A) and **array_limh**(B). Those elements of the result that are not within the range of either A or B are set to **miss_elt**. Those that are within the range of one argument are set to the corresponding element of that argument. Those that are within the range of both are set to the corresponding element of A if the corresponding element of B is **miss_elt**, to the corresponding element of B if the corresponding element of A is **miss_elt**, and to **miss_elt** otherwise. This operation is intended to be used to merge partially defined arrays, such as an array with only even elements defined (the others being **miss_elt**) and an array with only odd elements defined.

5.8 Record operations

The operations for a record type specified as T = **record**[$N_1 : T_1 ; \ldots ; N_k : T_k$] are the following. $N_1 \ldots N_k$ are the field names, and $T_1 \ldots T_k$ are the corresponding types.

operation	notation	functionality
create	**record**[$N_1 : V_1 ; \ldots ; N_k : V_k$]	$T_1, \ldots, T_k \rightarrow T$
$select_i$, $1 \leq i \leq k$	R . N_i	$T \rightarrow T_i$
$replace_i$, $1 \leq i \leq k$	R **replace** [N_i : V]	$T, T_i \rightarrow T$

test for **undef**	**is undef**(R)	T → <u>bool</u>
test for **miss_elt**	**is miss_elt**(R)	T → <u>bool</u>
test for **undef** or **miss_elt**	**is error**(R)	T → <u>bool</u>

Create **record**[$N_1 : V_1 ; \ldots ; N_k : V_k$]

This builds a record value { $(N_1, V_1), \ldots, (N_k, V_k)$ }. All of the field names associated with the type of the record being constructed must appear in the list, though some may appear with error values such as **undef**[T_i] or **miss_elt**[T_i].

Select R . N

This returns the value of the named field, that is, V_i if $N = N_i$.

Replace R **replace** [N : V]

This returns a record similar to R except that the N-field is changed to V.

Abbreviated notations are provided for compound selectors and multiple values in **replace** operations. See Section 6.5.

5.9 Operations for union types

The basic operations for a union type specified as T = **oneof**[$N_1 : T_1 ; \ldots ; N_k : T_k$] are a create operation and a test of a tag. The **tagcase** control structure explained in Section 7.3 is the mechanism for accessing constituent values from a value of union type. In the following, $N_1 \ldots N_k$ are the tag names, and $T_1 \ldots T_k$ are the corresponding constituent types.

operation	notation	functionality
create$_i$, $1 \le i \le k$	**make** T [N_i : V]	T_i → T
tag test$_i$, $1 \le i \le k$	**is** N_i (U)	T → <u>bool</u>

test for **undef**	**is undef**(U)	T → bool
test for **miss_elt**	**is miss_elt**(U)	T → bool
test for **undef** or **miss_elt**	**is error**(U)	T → bool

The operations **make** T [N : V] and **is** N (U) are type-correct only if N is a tag name of the type T and V is of that constituent type. The result of **make** T [N_i : V] is the pair (N_i : V) for any element V of T_i. The result of **is** N_i (U) is **true** if U = (N_i, anything), **undef**[boolean] if U is **undef**[T] or **miss_elt**[T], or **false** otherwise.

5.10 Type conversion operations

Type conversion operations are provided between integers and reals and between integers and characters.

operation	notation	functionality
real-to-integer	**integer**(X)	real → int
integer-to-real	**real**(I)	int → real
character-to-integer	**integer**(C)	char → int
integer-to-character	**character**(I)	int → char

In each case an argument value of **undef** or **miss_elt** yields the result **undef**. For other values the conversions act as follows:

integer(X):

If X is larger in magnitude than is representable as a proper element of **integer**, the result is **pos_over** or **neg_over**. If X is **zero_divide**, **pos_over**, **neg_over**, or **unknown**, the result is **undef**. If X is **pos_under** or **neg_under**, the result is zero. Otherwise, the result is obtained by rounding nonintegral values of X toward zero.

real(J):

All proper values of J are converted to the corresponding reals. If J is **zero_divide**, **pos_over**, **neg_over**, or **unknown**, the result is **undef**.

integer(C):

This operation yields the ASCII code for the character C.

character(J):

This operation is the inverse of **integer(C)**. Its result for values not in the range of **integer(C)** is not specified.

5.11 Type correctness of operations

In VAL the type of value produced by each expression can be determined by the translator from the properties of the operations as specified in this section. An operation in a program is type correct if and only if the types of its argument expressions are the same as the argument types specified for the operation. Note that for each operator the types of the results are determined when the types of the arguments are known.

6. CONSTANTS, VALUE NAMES, AND EXPRESSIONS

An expression is the basic syntactic unit denoting a tuple of values of some types. The *arity* of an expression is the size of the tuple of values it denotes. Two expressions are said to *conform* if they have the same arity and the corresponding values are of the same type. The design of the VAL language is such that the arity and types of an expression, and hence the conformity of two expressions, may be determined by inspection of the program. The simplest type of expression of arity one is a constant, a value name, or an operation applied to other expressions of arity one. The simplest type of expression of higher arity is a series of expressions of arity one separated by commas.

6.1 Constants

A constant is a syntactic unit of arity one whose value and type are manifest from its form.

Syntax:

```
constant ::= nil | true | false
        | integer-number | real-number | character-constant | character-string-constant
        | array_empty[type-spec]
        | undef[type-spec] | miss_elt[type-spec]
        | pos_over[type-spec] | neg_over[type-spec]
        | pos_under[type-spec] | neg_under[type-spec]
        | unknown[type-spec] | zero_divide[type-spec]
```

The values **undef**[type-spec] and **miss_elt**[type-spec] are constants denoting the undefined value and missing array element value of the type indicated in the type-spec. For example, **undef[array[integer]]** denotes the undefined value of type **array[integer]**. These two constants exist for all types, including array, record, and union types. The remaining constants for each data type are as follows:

The only constant of the **null** type is the reserved word **nil**.

The constants of the **boolean** type are the reserved words **true** and **false**.

The principal constants of the **integer** and **real** type are integer numbers and real numbers, the format of which are given in Section 3. There are also the following arithmetic error constants:

pos_over[integer]	**pos_over[real]**
neg_over[integer]	**neg_over[real]**
	pos_under[real]
	neg_under[real]
unknown[integer]	**unknown[real]**
zero_divide[integer]	**zero_divide[real]**

The constants of the **character** type are the characters enclosed in single quotes.

A character string enclosed in double quotes is a constant of type **array[character]** containing the individual characters of the string as elements. The first character is at index one.

The array constant **array_empty**[type] denotes the array of the indicated element type whose range is (1, 0), and hence has no elements.

There are no other array, record, or union constants, but various constructing operators may be used with constant arguments to denote "constant" arrays, records, or union elements.

Examples:

[1 : 1, 2, 3, 4, 5]	(array constant, see Section 6.4)
record [A : 6 ; B : 7.3]	(record constant)
make T [A : 6]	(constant of union type T)

6.2 Value names

A value name is a name which denotes a single computed value of a specific type. Every value name is introduced either in the header of a function definition (if the value name is a formal argument of the function being defined) or in a program construct such as a **let** block or a **for** block. In either case, each value name has a *scope* and a type, and has a unique value of that type for each instantiation during execution of the function or block with which the value name is

associated. The scope of a value name is the region of program text in which a reference to the value name denotes its value. The scope and type of any value name may be determined by inspection of the program construct that introduces it. Its value of course depends on the values present during the particular instantiation of the function or block.

The scope of a value name introduced as a formal argument of a function is the entire function definition, less any inner scopes that re-introduce the same value name. The type of such a value name is given by a type declaration in the function header. Its value is the value of the corresponding argument for the relevant invocation of the function.

Example:

```
function F ( X : integer returns real )
  <expression>
endfun
```

An appearance of value name X in the expression denotes the value of the argument with which F was invoked. Its type is **integer**.

The scope of a value name introduced in a program construct such as a *let* or *for* block is some region of the construct that depends on the nature of the construct, less any inner scopes that re-introduce the same value name. The manner in which the type and value of the value name are established depends on the form of the construct.

Example:

```
let
  X : real := 3.0 ;
  <another decldef> ;
  <another decldef> ;
  <another decldef> ;
in <expression>
endlet
```

The scope of X is the entire block, including the expression after **in**, less any inner scopes that re-introduce X. Its type is **real**; its value is 3.0. The **let** construct is described in Section 7.2. If this block had appeared within the scope of X introduced by some outer construct, that outer scope, with its value and type, would disappear within this **let** block.

6.3 Expressions

Expressions are built out of smaller expressions by means of operation symbols.

Syntax:

```
expression ::= level-1-exp | expression , level-1-exp        (the arities are added)
```

In the next 8 lines, the operators may only be used if all operands are of arity one.

```
level-1-exp ::= level-2-exp | level-1-exp | level-2-exp        (boolean "or")
level-2-exp ::= level-3-exp | level-2-exp & level-3-exp        (boolean "and")
level-3-exp ::= level-4-exp | ~ level-4-exp                    (boolean "not")
level-4-exp ::= level-5-exp | level-4-exp relational-op level-5-exp
level-5-exp ::= level-6-exp | level-5-exp || level-6-exp       (array concatenate)
level-6-exp ::= level-7-exp | level-6-exp adding-op level-7-exp
level-7-exp ::= level-8-exp | level-7-exp multiplying-op level-8-exp
level-8-exp ::= primary | unary-op primary

relational-op ::= < | <= | > | >= | = | ~=
adding-op ::= + | -
multiplying-op ::= * | /
unary-op ::= + | -

primary ::= constant | value-name          (these have arity one)
          | (expression)                  (same arity as expression in parentheses)
          | invocation                    (arity is the number of values returned)
          | array-ref | array-generator
          | record-ref | record-generator (These eight forms
          | oneof-test | oneof-generator  have arity one.)
          | error-test | prefix-operation
```

```
        | conditional-exp
        | let-in-exp                  (These five structures are
        | tagcase-exp                  described in Section 7.
        | iteration-exp                They have arbitrary arity.)
        | forall-exp
value-name ::= name
```

In an invocation, the arity of the expression in parentheses must be equal to the number of arguments required by the function.

```
invocation ::= function-name (expression)
function-name ::= name
array-ref ::= primary [expression]
array-generator ::= [ expression : expression { ; expression : expression } ]
        | primary [ expression : expression { ; expression : expression } ]
record-ref ::= primary . field-name
```

In the next 7 forms, all expressions must have arity one except as otherwise noted, and the resultant expressions always have arity one.

```
record-generator ::= record [ field-name : expression { ; field-name : expression } ]
        | primary replace [ field : expression { ; field : expression } ]
field ::= field-name { . field-name }
field-name ::= name
oneof-test ::= is tag-name (expression)
oneof-generator ::= make type-spec [ tag-name : expression ]
tag-name ::= name
error-test ::= is undef (expression) | is miss_elt (expression)
        | is error (expression) | is zero_divide (expression)
        | is pos_over (expression) | is neg_over (expression)
        | is pos_under (expression) | is neg_under (expression)
        | is over (expression) | is under (expression)
        | is arith_error (expression) | is unknown (expression)
```

The arities of the argument expressions for a prefix operation are as shown, and the result arity is always one.

prefix-operation ::= **integer** (expression)	(arity = 1)
| **real** (expression)	(arity = 1)
| **character** (expression)	(arity = 1)
| **abs** (expression)	(arity = 1)
| **exp** (expression)	(arity = 1)
| **mod** (expression)	(arity = 2)
| **max** (expression)	(any arity)
| **min** (expression)	(any arity)
| **array_fill** (expression)	(arity = 3)
| **array_limh** (expression)	(arity = 1)
| **array_liml** (expression)	(arity = 1)
| **array_size** (expression)	(arity = 1)
| **array_adjust** (expression)	(arity = 3)
| **array_addh** (expression)	(arity = 2)
| **array_addl** (expression)	(arity = 2)
| **array_remh** (expression)	(arity = 1)
| **array_reml** (expression)	(arity = 1)
| **array_join** (expression)	(arity = 2)
| **array_setl** (expression)	(arity = 2)

Note that operators obey the customary precedence rules: unary plus and minus have highest priority; multiplicative operators (*, /) are next; additive operators (+, -) are next; "||" is next; relational operators (<, <=, >, >=, =, ~=) are next; "~" is next; "&" is next; and "|" has lowest priority.

Examples of expressions of arity one:

```
A
true
3.7E-02
"X"
"XYZ" || [1 : C] || "PQR"
array_empty[integer]
zero_divide[real]
X > 2 & Z < 3 * K
- X + 3 * B
3 * ( X + Y )
func(3+X, Y)                          (if "func" returns one value)
[ 3 : Z ]
A [ 3 : Z ]
A [ 4, J ]
R . X . Y . ZZ
record [ A : P ; B : Q ]
R replace [ A.X : P ; B.Y : Q ]
is A (U)
make T [A] : 3
is over (X)
undef[real]
if P then 4 else 5 endif              (see Section 7)
```

5.4 Abbreviations for array operations

The syntax provides abbreviated forms for the *select*, *append*, and *create by elements* operations, to allow convenient array creation and handling of multi-dimensional arrays.

Since multi-dimensional arrays are represented as arrays of arrays, the straightforward way to select an element is with an expression such as

```
A[J][K][L]
```

This may be written

```
A[J, K, L]
```

The expression within brackets has arity three.

The *append* operation can be used for multi-dimensional arrays by using an expression of arity greater than one for the subscripts. Thus

A [J, K, L : V]

is equivalent to

A [J : A[J] [K : A[J, K] [L : V]]]

that is, A with its J, K, L element replaced by V.

Several values may be appended at consecutive indices by using an expression of arity greater than one.

A [J : V, W, X]

is equivalent to

A [J:V ; J+1:W ; J+2:X]

If multi-dimensional arrays are being used, the last index is the one that varies when multiple data items are present.

A [J, K, L : V, W, X]

is equivalent to

A [J, K, L : V ; J, K, L+1 : W ; J, K, L+2 : X]

These expressions need not be constructed by listing expressions of arity one separated by commas. Other forms of expressions with high arity will be described in Section 6.6. For example:

A [J : TRIPLE(X, Y, Z)]

fills in indices J, J+1, and J+2 if TRIPLE is a function returning three values.

Finally, *append* operations may be composed by writing the J : V pairs in sequence within the brackets, separated by semicolons.

A [J_1 : V_1 ; J_2 : V_2 ; . . . ; J_N : V_N]

is equivalent to

A [J_1 : V_1][J_2 : V_2] . . . [J_N : V_N]

where, as noted above, J_i and/or V_i may be expressions of any arity.

All of the abbreviations permissible for the *append* operation are permissible for the *create by elements* operation.

Examples:

[3 : X ; 5 : Y, Z]

is an array with range (3, 6), and elements X, miss_all, Y, and Z.

[1 : A]

is a "singleton" array with low and high indices both one.

6.5 Abbreviations for record operations

There are abbreviated forms for the **replace** operation to allow convenient handling of compound selectors and multiple data elements.

Accessing records with compound selectors is performed in the straightforward way:

R.A.B.C

Compound selectors may be used in **replace** operations by writing the field names separated by periods:

R **replace** [A.B.C : V]

is equivalent to

R **replace** [A : R.A **replace** [B : R.A.B **replace** [C : V]]]

that is, R with its A.B.C subcomponent replaced by V.

replace operations may be composed by writing the N : V pairs in sequence within the brackets, separated by semicolons.

R **replace** [A : V ; B : W ; C.D : X]

is equivalent to

((R **replace** [A : V]) **replace** [B : W]) **replace** [C.D : X]

6.6 Expressions of higher arity

The program structures provided in VAL for conditional computation and iteration are expressions of arbitrary arity, and are described in Section 7. Such expressions, or function invocations, may occur in program text in places that require a tuple of values of specified types: the argument list of an operation or function invocation, the body of a function definition, a list of array indices or elements in an array operation, or in building the program structures presented in Section 7.

6.7 Function invocations

A function invocation consists of the name of the function followed by an argument list within parentheses. (The syntax is the same for internal and external and external functions.) The argument list is an expression, whose arity and types conform to the arguments required by the function. This information is given in the header of the function definition. See Section 8. The argument list is usually written as a series of expressions of arity one separated by commas, but it may be any expression.

A function invocation is itself an expression whose arity and types are the number and types of the values returned by the function, which information also appears in the function's header. An invocation that returns one value may appear in expressions with complete generality, such as an argument to arithmetic, array, and record operations. An invocation that returns several values may only be used where expressions of higher arity are permitted.

In the following examples, SINGLE, DOUBLE, and TRIPLE each take 3 arguments and return 1, 2, or 3 values, respectively:

```
K := 3 + Z * SINGLE (X + 1, 3, SINGLE (X + 2, 4, W)) ;
```

In the following example, if P is **false**, F and G are defined to be DOUBLE (X, Y, Z), while H is defined to be W:

```
F, G, H := if P then TRIPLE (X, Y, Z) else DOUBLE (X, Y, Z), W endif ;
```

Since the argument list for any function may be any expression, it may be a multiple-result function invocation or other program structure.

```
3 + SINGLE (TRIPLE (X, Y, Z))

3 + SINGLE (P, DOUBLE (X, Y, Z))

4 + SINGLE (if P then 4, 5 else DOUBLE (P, Q, R) endif, X)
```

The last example invokes SINGLE with three arguments, of which the first two are either 4 and 5 or the two values returned by DOUBLE. The third argument to SINGLE is always X.

7. PROGRAM STRUCTURES

The program structures described in this section are specific forms of expressions. If their arity is one, they may appear in arithmetic operations.

Example:

```
if P then X else Y endif + 3
```

This expression has value X+3 or Y+3, depending on P.

7.1 The IF construct

The conditional expression selects one of several expressions, depending on the values of boolean expressions.

Syntax:

```
conditional-exp ::= if expression then expression
                    { elseif expression then expression }
                    else expression
                    endif
```

The expressions following **if** and **elseif** are *test expressions*. Their arity must be one and their type **boolean**. The expressions following **then** and **else** are the *arms*. They must conform to each other, and the entire construct conforms to the arms.

The entire construct is an expression whose tuple of values is that of the first arm whose test expression is true, or the final arm if all test expressions are false. If any test expression needed to evaluate the construct is an error value (**undef[boolean]** or **miss_elt[boolean]**), the value of the entire construct is a tuple of **undef** values of the appropriate types. (If a test expression has value **true**, later test expressions are not needed and may have error values without affecting evaluation of the construct).

The **if** construct introduces no value names. All value name scopes pass into an **if** construct. If the scope of a value name includes an **if** construct, it includes all of the expressions of that construct, so that value name may be used anywhere inside the conditional construct.

7.2 The LET construct

The purpose of this construct is to introduce one or more value names, define their values, and evaluate an expression within their scope (that is, making use of their defined values).

Syntax:

```
let-in-exp ::=
        let decldef-part
        in expression
        endlet
decldef-part ::= decldef { ; decldef } [ ; ]
decldef ::= decl
        | def
        | decl { , decl } := expression
decl ::= value-name { , value-name } : type-spec
def ::= value-name { , value-name } := expression
```

Every value name introduced in a **let** block must be declared exactly once and defined exactly once in that block. The declaration may be part of the definition, or it may be by itself preceding the definition.

Examples:

```
X : integer ;                    (declaration)
X := 3 ;                         (definition)
Y : real := 4.7 + Q ;            (declaration as part of definition)
```

The declaration of a value name must precede or be part of its definition. Each value name must be defined before it is used (on the right hand side of another definition). Declarations and definitions may be mixed in any order as long as these requirements are met.

Several value names may be declared at once:

```
X, Y, Z : real ;
```

This declares all 3 names to be **real**.

Several value names may be defined at once. The number and types of the names must conform to the arity and types of the expression on the right hand side.

```
X, Y, Z := 1.0, 2.0, 3.0 ;
P, Q, R := TRIPLE(X, Y, Z) ;
```

Several value names may be declared and defined at once. In this case, each of a group of value name names preceding a type specification are declared to be of that type.

```
X : integer, Y, Z : real := 3, 4.0, 5.0 ;
```

This declares X to be integer, and both Y and Z to be real.

The declarations, definitions, and combined declarations and definitions are separated by semicolons; a semicolon after the last is optional.

The scope of each value name introduced in a **let** block is the entire block less any inner constructs that re-introduce the same value name. However, a value name must not be used in the definitions preceding its own definition.

All scopes for value names not introduced in a given **let** block pass into that block. Hence, if the scope of a value name (introduced by an outer construct) includes a **let** block and that value name is not re-introduced, it may be referred to freely within the block.

Example:

```
let X : real ; T : real ;
  T := P + 3.7 ;
  X := T + 2.4 ;
in X * T
endlet
```

In this example, the value of P is imported from the outer context. The scopes of T and X are both the entire block. A reference to X in the definition of T would be illegal because it is within the scope of X but does not follow the definition of X. The arity of this construct is one, and its type is real, because X*T has arity one and type **real**.

Since a value name may not be used until after it has been defined, and must be defined only once in a block, it may not appear in its own definition. Hence definitions such as

```
I := I + 1 ;
```

are never legal in **let** blocks (though they may occur in **iter** clauses of **for** blocks; see Section 7.4.)

The expression following the word **in** is in the scope of all of the introduced value names, and hence can make use of their definitions. The entire **let** construct conforms to this expression.

7.3 The TAGCASE construct

This selects one of a number of expressions, depending on the tag of a **oneof** value, and extracts the constituent value.

Syntax:

```
tagcase-exp ::=
        tagcase [ value-name := ] expression [ ; ]
        tag-list : expression
        { tag-list : expression }
        [ otherwise : expression ]
        endtag
tag-list ::= tag tag-name { , tag-name }
```

The entire construct is an expression whose values are those of the expression in the arm whose tag name matches that of the value of the test expression. If no match is found, the arm following the word **otherwise** is used. All arms must conform to each other, and the entire construct conforms to the arms.

The expression following the word **tagcase** must be of arity one and of a **oneof** type. The tag names appearing in the arms of the construct must be tags of that **oneof** type. If they comprise all the tags of that type, the **otherwise** arm is not used; if not, the **otherwise** arm is required.

If a value name and ":=" appear after the word **tagcase**, that name is introduced for each arm of the construct except the **otherwise** arm. Its scope in each case is the expression in that arm, and its type is the constituent type indicated by the tag name for that arm. If an arm is evaluated (meaning that the tag of the test expression matches the tag name of the arm), the value name is defined to be the constituent value from the test expression. If the value name and ":=" do not appear, the constituent value is not made available inside the arms.

Example:

Let X be of type

```
oneof [ A : integer ; B : array[integer] ; C : real ; D : boolean ]
```

If X has tag A and constituent value 3,

```
tagcase P := X ;
  tag A : P + 4
  tag B : P[6]
  otherwise : 5
endtag
```

has value 7. The first arm is taken, and P (whose type is **integer** in that arm) is defined to be 3, the constituent value of X. If X has tag B and constituent value some array whose sixth element is 2, the value of the above construct is 2. In that case, P is defined to be the array. If X has tag C or D, the construct has value 5. In that case the constituent value is not available, since the value name's scopes do not include the **otherwise** arm. (This is because the **otherwise** arm can encompass different constituent types, so the type of the value name could not be determined.)

More than one tag name may share the same arm if they indicate the same type. In this case, the tag names are all listed, separated by commas, after the word **tag**.

Example:

Let X be of type

```
oneof [ A : integer ; B : real ; C : integer ]
```

Then the following is permissible:

```
tagcase P := X ;
  tag A, C : expression1          (P is integer here)
  tag B : expression2             (P is real here)
endtag
```

All scopes of value names other than the one appearing after the word **tagcase** pass into the construct. An outer scope for a value name with the same name as the one appearing after the word **tagcase** does not pass into the **tagcase** construct.

If the value of the test expression is an error value, the value of the entire construct is a tuple of **undef** values of the appropriate types.

7.4 The FOR construct

This performs sequential iteration in which one iteration cycle depends on the results of previous cycles. The construct introduces a number of value names, called *loop* names, which convey information from one cycle to the next.

Syntax:

```
iteration-exp ::=
        for decldef-part
        do iter-end
        endfor
iter-end ::= if expression then iter-end
                { elseif expression then iter-end }
                else iter-end endif
        | tagcase [ value-name := ] expression [ ; ]
                tag-list : iter-end
                { tag-list : iter-end }
                [ otherwise : iter-end ] endtag
        | let decldef-part in iter-end endlet
        | expression
        | iter def-part enditer
def-part ::= def { ; def } [ ; ]
```

The loop names are those appearing in the declarations and definitions following the word **for**. These declarations and definitions have the same form as in a **let** block.

The behavior of the **for** construct is as follows: The loop names are initialized, only once, to the values indicated in the definitions appearing after the word **for**, and the first iteration cycle begins. During each iteration cycle these names have fixed values.

The iteration body is then evaluated, using the current definitions of the loop names. The result of that evaluation is either a decision to terminate the iteration, with values to be returned, or a decision to iterate again with new definitions for the loop names.

The iteration body consists of an **if** construct, **tagcase** construct, or a tree of **if**, **tagcase**, and **let** constructs, with a slight modification: the arms may either be conventional expressions, or may consist of **iter**, some redefinitions, and **enditer**. There may be many arms of each type.

If the arm that is chosen for evaluation is an expression, the iteration terminates, and the values of the expression are the values of the entire **for** construct. All such arms must conform to each other, and the entire construct conforms to these arms.

If the chosen arm consists of **iter**, some redefinitions, and **enditer**, those loop names are redefined according to the the right hand sides of the redefinitions, and evaluation of the body is repeated.

Examples:

```
for Y : integer := 1 ; P : integer := N ;
do      if P ~= 1 then iter Y := Y*P ; P := P-1 ; enditer
        else Y
        endif
endfor
```

This computes the factorial of N. It introduces loop names Y and P, which are both integer. Their initial values are 1 and N, respectively.

The body of this construct is an **if/then/else** construct whose first arm is a redefinition and whose second arm is the expression "Y". Accordingly, at the beginning of each iteration cycle P is tested. If it is not one, the **iter** arm gives Y the new value Y*P and P the new value P-1, and another cycle begins. If P is one, the iteration terminates with the value Y.

```
for T : real := X ;
do      let D : real := (X/T - T)/2 ;
        in      if D < eps then T
                else iter T := T + D ; enditer
                endif
        endlet
endfor
```

This computes the square root of X, using Newton's method. The iteration body uses a **let** block to introduce the temporary name D. It imports the value *eps* from the context in which the block appears.

The next example reverses the list given as "INPUT", by initially defining T to be INPUT and U to be the empty list, and then repeatedly moving items from T to U. Assume the type LIST has been defined by

```
type LIST = oneof [ empty : null ; nonempty : record [ item : real ; rest : LIST ]]
```

A "LIST" is a chain of records containing an arbitrary number (perhaps zero) of reals.

```
for T, U : LIST := INPUT, make LIST [ empty : nil ] ;
do      tagcase Z := T ;
        tag empty : U
        tag nonempty :
                iter
                T, U := Z.rest, make LIST [ nonempty : record [ item : Z.item ; rest : U ]] ;
                enditer
        endtag
endfor
```

The loop value names must all be different. Their scopes are the entire **for** construct less any inner blocks that re-introduce the same name. They are declared and initially defined in the same manner as in a **let** block. As in a **let** block, each name must be declared exactly once and defined exactly once, and may appear on the right hand side only in definitions after its own. Each declaration, definition, or combined declaration and definition must be followed by a semicolon except the last, for which the semicolon is optional.

Within each **iter** arm the redefined value names must be a subset of the loop names. These redefinitions may make use of the previous values of all names, including the one being redefined. These redefinitions do not include declarations, since the types of the loop names were declared at the beginning of the **for** construct. Each redefinition must be followed by a semicolon except the last, for which the semicolon is optional.

Unlike the definitions in a **let/in** block or the initial loop value definitions in a **for** block, a redefinition in an **iter** clause may contain, on its right hand side, loop names that appear on the left hand side of the same or later redefinitions. In such a case, the "old" value is used, that is, the value that the name had on the iteration cycle just ending. If the name appeared on the left hand side of an earlier redefinition, its "new" value is used, that is, the result of that redefinition.

Hence a redefinition such as

```
J := J + 1 ;
```

is legal, and means that the next iteration cycle is to begin with a value of J which is one greater than its value on the cycle just ended. In the factorial example given above, the iteration clause

```
iter Y := Y*P ; P := P-1 ; enditer
```

multiplies Y by the *old* value of P. If the order had been reversed:

```
iter P := P-1 ; Y := Y*P ; enditer
```

Y would be multiplied by the *new* value of P, and the example program would compute the factorial of N-1.

The simplest way to redefine two or more loop variables in terms of each others' old values is to use a multiple assignment. For example:

```
iter X, Y := Y, X ; enditer
```

exchanges the values of X and Y for the next iteration cycle.

A loop name not appearing in a redefinition after **iter** retains its old value.

The scopes of any value names other than the loop names pass from outer blocks into the **for** block.

If an error occurs while determining which arm of the loop body to evaluate (a test expression in an **if** or **tagcase** construct evaluates to an error value) the iteration terminates and returns as its value a tuple of **undef** values of appropriate types. An error arising elsewhere in the loop body does not cause special action: If it arises in an **for** arm the indicated loop name is defined to be the error value and the iteration continues. If it arises in an expression giving the final value to be returned, that error value simply appears in the result.

7.5 The FORALL construct

This generates one or more sets of values, of uniform type within each set, and either returns them as arrays or returns the result of some operation (such as addition) on them. The former case is indicated by the word **construct**, the latter by the word **eval** followed by the name of the operator. The values may not depend on each other -- the intention is that they be computed simultaneously on a parallel computer capable of doing so.

This construct introduces one or more *index* value names of type integer and a number of optional temporary value names, the latter in the the same manner as in a **let** block.

Syntax:

forall-exp ::=
    forall value-name **in** [expression] { , value-name **in** [expression] }
    [decldef-part]
    forall-body-part
    { forall-body-part }
    endall

forall-body-part ::= **construct** expression | **eval** forall-op expression

forall-op ::= **plus** | **times** | **min** | **max** | **or** | **and**

The index names are those appearing before the word **in**. The temporary names are those appearing in the declarations and definitions.

The index and temporary names must all be different. Their scopes are the entire construct less any inner blocks that re-introduce the same value name. The types of the indices are integer. The types of the temporary names are specified in their declarations. As in a **let** expression, a temporary name may not appear in definitions preceding its own.

Each expression appearing in brackets after the word **in** is of arity two with both types integer. The two components are the low and high limits, inclusive, for the index. For each number within those limits, the index is defined to be that number, the definitions of the temporary names are made, and all the parts are evaluated. When more than one index is given, this is done for each point in the "Cartesian product" of the ranges, that is, for every combination of index values.

In a **construct** part, the expression is evaluated for each index value, and for each component of the expression, an array is formed having the same limits as the limits given for the index and elements equal to the values obtained. If more than one index is given, a multidimensional array is formed, that is, an array of arrays, with the first index referring to the outermost array. If some component of the expression is an error value for some index value, that array element is simply set to that error value.

Example:

```
forall J in [ 1, 4 ]
X : real := square_root(real(J));
construct J, X, X+1.0
endall
```

creates 3 arrays, all with range 1 to 4. The first is integer and contains values 1, 2, 3, and 4. The second is real and contains 1.0, 1.414, 1.732, and 2.0. The last is real and contains 2.0, 2.414, 2.732, and 3.0. This **forall** block is an expression of arity three whose values are these three arrays.

```
forall J in [ A, B ], K in [ C, D ]
construct <expression>
endall
```

is equivalent to

```
forall J in [ A, B ]
construct
        forall K in [ C, D ]
        construct <expression>
        endall
endall
```

and constructs a two-dimensional array, that is, an array whose limits are [A, B] and whose elements are arrays whose limits are [C, D].

In an **eval** part, the operation must be one of **plus**, **times**, **min**, **max**, **or**, or **and**. The arity of the expression must be one, and its type must be appropriate for the operation: real or integer for **plus**, **times**, **min**, or **max**, boolean for **or** or **and**. The expression is evaluated for each index value, and the operation is performed on the collection of values that are produced. If multiple indices are used, the operation is performed on the entire collection of values produced for all combinations of index values.

Example:

```
forall J in [ 1, N ]
eval plus J*J
endall
```

returns $\sum_{j=1}^{N} j^2$.

The result of an entire **forall** block is an expression constructed by concatenating the results of all of the parts.

Example:

```
forall J in [ 1, N ]
X : real := square_root(real(J)) ;
eval plus J*J
construct J, X, X+1.0
endall
```

is an expression of arity 4 and types **integer**, **array[integer]**, **array[real]**, and **array[real]**.

If one of the bounds is an error value, or the lower bound is greater than the upper bound plus one, the result of the entire **forall** block is a tuple of **undef** values of appropriate types. If the lower bound is equal to the upper bound plus one, the result of each **construct** part is an array with no elements, and the result of each **eval** part is 0, 1, **pos_over**, **neg_over**, **false**, or **true**, if the operator is **plus**, **times**, **min**, **max**, **or**, or **and**, respectively.

The scopes of any value names other than the index and temporary names, introduced in outer constructs, pass into the **forall** block.

8. FUNCTION DEFINITIONS

A VAL program consists of a collection of modules, each defining one "external" function and any number of "internal" functions. Each function is defined by a *function definition*, which is a piece of text consisting of:

(1) The word **function**.

(2) The function name and information specifying the arity and types of its arguments and returned values. This information is called the "header".

(3) The type definitions used in the function definition. If this is an external function, the declarations of other external functions used by this module appear here also.

(4) The definitions of the internal functions subsidiary to this one. Function definitions may thus be nested arbitrarily.

(5) The expression giving the values to be returned by the function. This is the "body" of the function definition.

(6) The word **endfun**.

The definition of an external function is an entire module in VAL. Definitions of internal functions appear within function definitions in item 4 of the above list.

Syntax:

```
module ::= external-function-def
external-function-def ::=
        function function-header
        [ type-external-def-part ]
        { internal-function-def }
        expression
        endfun
internal-function-def ::=
        function function-header
        [ type-def-part ]
        { internal-function-def }
        expression
        endfun
```

```
type-external-def-part ::= type-external-def { ; type-external-def } [ ; ]
type-external-def ::= type-def | external-def
type-def-part ::= type-def { ; type-def } [ ; ]
type-def ::= type type-name = type-spec
external-def ::= external function-header
function-header ::= function-name ( decl { ; decl } returns type-spec { , type-spec } )
function-name ::= name
```

Example:

```
function sum_of_squares (X, Y : real returns real)
  X*X + Y*Y
endfun
```

Only the external (outermost) function defined in a module is accessible to other modules.

Optional type definitions may appear after the header to give names to types. These user-defined names may be used anywhere in the function definition, including its own header. The type definitions (and **external** declarations) are separated from each other by semicolons; a semicolon after the last is optional.

Example:

```
function complex_multiply (X, Y : complex returns complex)
  type complex = record [ re, im : real ];
  record [ re : X.re * Y.re - X.im * Y.im ; im : X.im * Y.re + X.re * Y.im ]
endfun
```

8.1 The header and value transmission

The list of formal arguments and their type specifications appears in the header between the left parenthesis and the word **returns**. These declarations are separated from each other by semicolons. Each declaration may contain several value names, which are separated from each other by commas.

The scope of the formal arguments is the body of the function (the expression), less any inner constructs which re-introduce the same value name. Their types are as given in the header declarations, and their values are the values of the arguments given at function invocation. The types of the returned values are given in the list of type specifications, separated by commas, appearing after the word **returns**. This list of types must conform to the body. In every invocation of a function, the number and types of the arguments and returned values must match those of the definition.

The meaning of a function invocation is as follows: If the function F is defined by

```
function F ( a1 : t1 ... aN : tN returns s1 ... sK )
BODYEXP
endfun
```

then, assuming the definition is correct and conforms to its invocation, the invocation

```
F(ARGEXP)
```

is equivalent to

```
let a1 : t1 ... aN : tN := ARGEXP in BODYEXP endlet
```

8.2 The EXTERNAL declaration

All functions used in a module that are not defined in that module must be declared in an **external** declaration. This declaration consists of the word **external** followed by a copy of the function's header, which is used by the translator for type checking.

Example:

```
function tan ( X : real returns real )
external sin ( Q : real returns real ) ;
external cos ( Q : real returns real ) ;
sin(X) / cos(X)
endfun
```

This module defines the external function *tan*. Since it uses the functions *sin* and *cos*, which are not defined here, they must appear in **external** declarations. (They must be defined in other modules or accessed in a subroutine library.) The **external** declarations contain the headers for *sin* and *cos*, just as they might appear in the definitions of those two functions. The formal arguments appearing in the headers ("Q" in the preceding example) have no significance; they are

included only for syntactic consistency. The intention is that the headers be copied verbatim from the modules defining *sin* and *cos* into the module defining *tan*.

A module's **external** declarations must appear following the header of the outermost function definition of that module, even if the functions being declared are used only by internal functions. The **external** declarations may precede, follow, or be mixed with the type definitions of the outermost function definition.

8.3 Inheritance of data, type definitions, and external declarations

A function has access only to the data presented to it in its invocation. No data values are imported from any enclosing function definition. Type definitions made in one function definition are inherited by all functions subsidiary to it. A redefinition in an internal function of a type name already defined in an outer context is not permitted.

External declarations made in the outermost function definition are inherited by all internal functions.

8.4 Scope of function definitions

The scope of an external function definition consists of all modules of the program except the module defining the function. That is, any external function may be invoked from anywhere except in the module giving that function's definition. The scope of an internal function consists solely of the immediately enclosing function definition. Note that this precludes any recursion or mutual recursion.

The scope rules for functions and type definitions are illustrated by the following example:

```
function F ( <header> )
external FF ( <header> ) ;
type T = <type-spec> ;
  function G ( <header> )
  type U = <type-spec> ;
    function M ( <header> )
        function N ( <header> )
        BODY_N
        endfun
    BODY_M
    endfun
  BODY_G
  endfun
  function H ( <header> )
    function P ( <header> )
    BODY_P
    endfun
  BODY_H
  endfun
BODY_F
endfun
```

the body of	may invoke functions
F	FF (external), G, H (internal)
G	FF (external), M (internal)
M	FF (external), N (internal)
N	FF (external)
H	FF (external), P (internal)
P	FF (external)

the body and header of	may use defined types
F	T
G	T, U
M	T, U
N	T, U
H	T
P	T

The modules comprising a program are translated separately. The manner in which their names are used to access them in libraries and the manner in which they are linked into a complete program is dependent on the implementation. No recursive invocations among external or internal functions are permitted.

Appendix I - Formal Syntax

```
module ::= external-function-def
external-function-def ::=
        function function-header
        [ type-external-def-part ]
        { internal-function-def }
        expression
        endfun
internal-function-def ::=
        function function-header
        [ type-def-part ]
        { internal-function-def }
        expression
        endfun
type-external-def-part ::= type-external-def { ; type-external-def } [ ; ]
type-external-def ::= type-def | external-def
type-def-part ::= type-def { ; type-def } [ ; ]
type-def ::= type type-name = type-spec
external-def ::= external function-header
function-header ::= function-name ( decl { ; decl } returns type-spec { , type-spec } )
function-name ::= name
expression ::= level-1-exp | expression , level-1-exp
level-1-exp ::= level-2-exp | level-1-exp | level-2-exp
level-2-exp ::= level-3-exp | level-2-exp & level-3-exp
level-3-exp ::= level-4-exp | ~ level-4-exp
level-4-exp ::= level-5-exp | level-4-exp relational-op level-5-exp
level-5-exp ::= level-6-exp | level-5-exp || level-6-exp
level-6-exp ::= level-7-exp | level-6-exp adding-op level-7-exp
level-7-exp ::= level-8-exp | level-7-exp multiplying-op level-8-exp
level-8-exp ::= primary | unary-op primary
```

```
relational-op ::= < | <= | > | >= | = | ~=
adding-op ::= + | -
multiplying-op ::= * | /
unary-op ::= + | -

primary ::= constant | value-name
        | (expression)
        | invocation
        | array-ref | array-generator
        | record-ref | record-generator
        | oneof-test | oneof-generator
        | error-test | prefix-operation
        | conditional-exp
        | let-in-exp
        | tagcase-exp
        | iteration-exp
        | forall-exp
value-name ::= name
invocation ::= function-name (expression)
array-ref ::= primary [expression]
array-generator ::= [ expression : expression { ; expression : expression } ]
        | primary [ expression : expression { ; expression : expression } ]
record-ref ::= primary . field-name
record-generator ::= record [ field-name : expression { ; field-name : expression } ]
        | primary replace [ field : expression { ; field : expression } ]
field ::= field-name { . field-name }
field-name ::= name
oneof-test ::= is tag-name (expression)
oneof-generator ::= make type-spec [ tag-name : expression ]
tag-name ::= name
error-test ::= is undef (expression) | is miss_elt (expression)
        | is error (expression) | is zero_divide (expression)
```

| **is pos_over** (expression) | **is neg_over** (expression)

| **is pos_under** (expression) | **is neg_under** (expression)

| **is over** (expression) | **is under** (expression)

| **is arith_error** (expression) | **is unknown** (expression)

prefix-operation ::= **integer** (expression)

| **real** (expression)

| **character** (expression)

| **abs** (expression)

| **exp** (expression)

| **mod** (expression)

| **max** (expression)

| **min** (expression)

| **array_fill** (expression)

| **array_limh** (expression)

| **array_liml** (expression)

| **array_size** (expression)

| **array_adjust** (expression)

| **array_addh** (expression)

| **array_addl** (expression)

| **array_remh** (expression)

| **array_reml** (expression)

| **array_join** (expression)

| **array_setl** (expression)

constant ::= **nil** | **true** | **false**

| integer-number | real-number | character-constant | character-string-constant

| **array_empty**[type-spec]

| **undef**[type-spec] | **miss_elt**[type-spec]

| **pos_over**[type-spec] | **neg_over**[type-spec]

| **pos_under**[type-spec] | **neg_under**[type-spec]

| **unknown**[type-spec] | **zero_divide**[type-spec]

type-spec ::= basic-type-spec

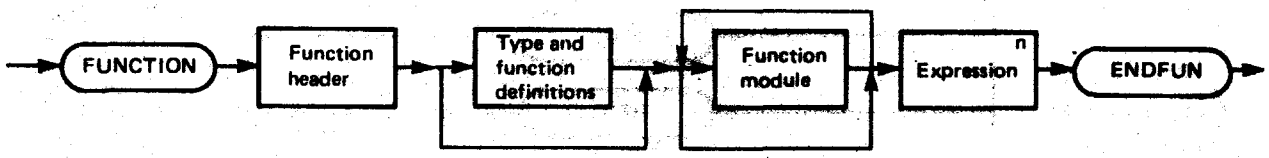
| compound-type-spec

```
        | type-name
basic-type-spec ::= null | boolean | integer | real | character
compound-type-spec ::= array [type-spec]
        | record [ field-spec { ; field-spec } ]
        | oneof [ tag-spec { ; tag-spec } ]
field-spec ::= field-name { , field-name } : type-spec
tag-spec ::= tag-name { , tag-name } [ : type-spec ]
type-name ::= name
conditional-exp ::= if expression then expression
                { elseif expression then expression }
                else expression
                endif
let-in-exp ::=
        let decldef-part
        in expression
        endlet
decldef-part ::= decldef { ; decldef } [ ; ]
decldef ::= decl
        | def
        | decl { , decl } := expression
decl ::= value-name { , value-name } : type-spec
def ::= value-name { , value-name } := expression
tagcase-exp ::=
        tagcase [ value-name := ] expression [ ; ]
        tag-list : expression
        { tag-list : expression }
        [ otherwise : expression ]
        endtag
tag-list ::= tag tag-name { , tag-name }
iteration-exp ::=
        for decldef-part
        do iter-end
```

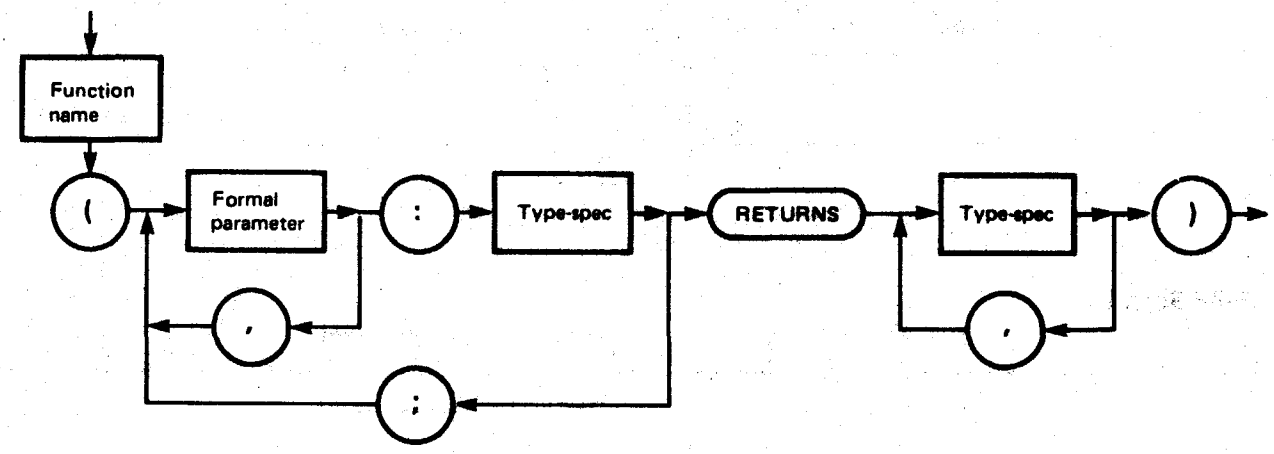
```
        endfor
iter-end ::= if expression then iter-end
                { elseif expression then iter-end }
                else iter-end endif
        | tagcase [ value-name := ] expression [ ; ]
                tag-list : iter-end
                { tag-list : iter-end }
                [ otherwise : iter-end ] endtag
        | let decldef-part in iter-end endlet
        | expression
        | iter def-part enditer
def-part ::= def { ; def } [ ; ]
forall-exp ::=
        forall value-name in [ expression ] { , value-name in [ expression ] }
        [ decldef-part ]
        forall-body-part
        { forall-body-part }
        endall
forall-body-part ::= construct expression | eval forall-op expression
forall-op ::= plus | times | min | max | or | and
```

VAL Syntax Charts

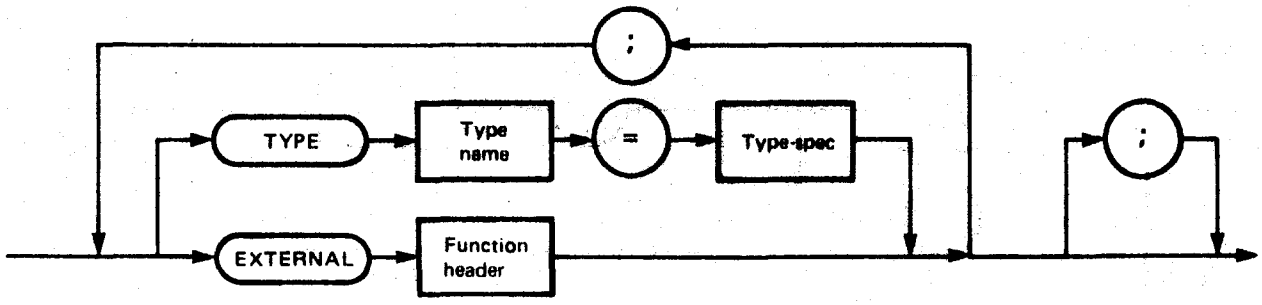
Function Module



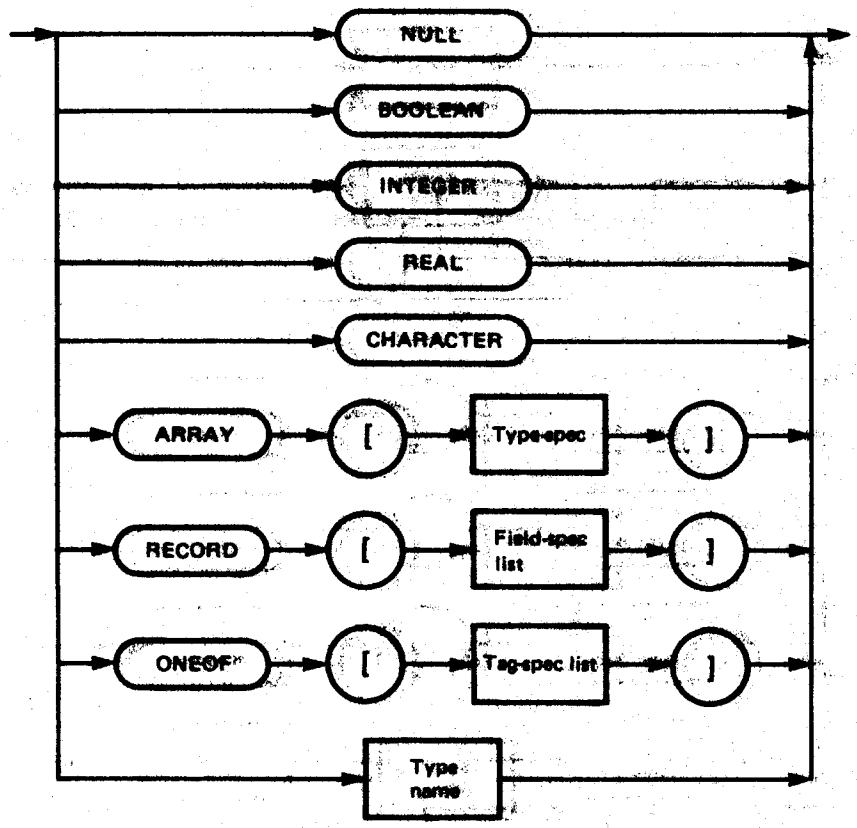
Function Header



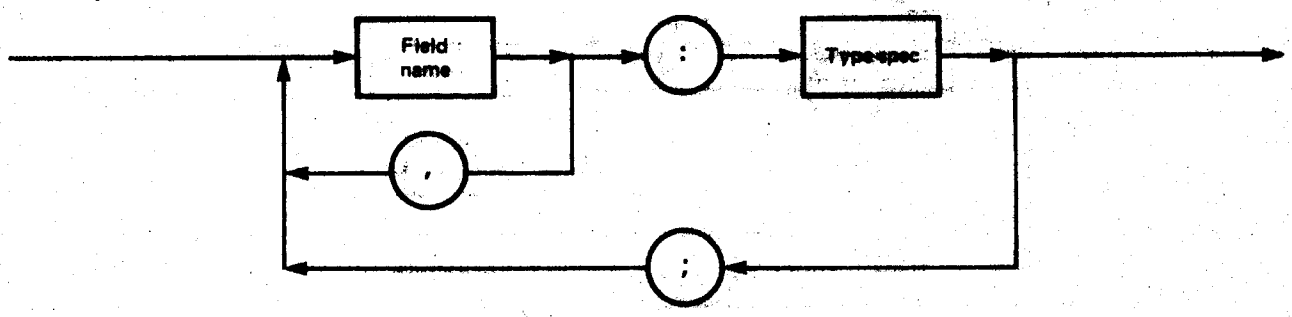
Type and Function Definitions



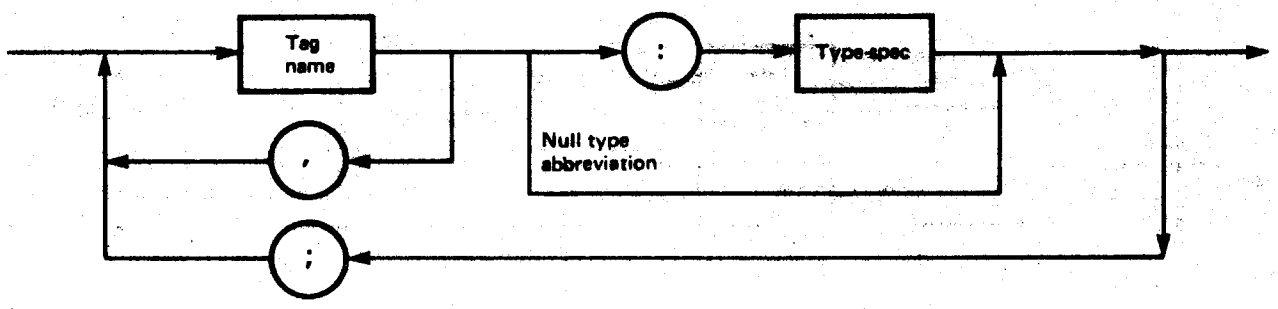
Type-Spec



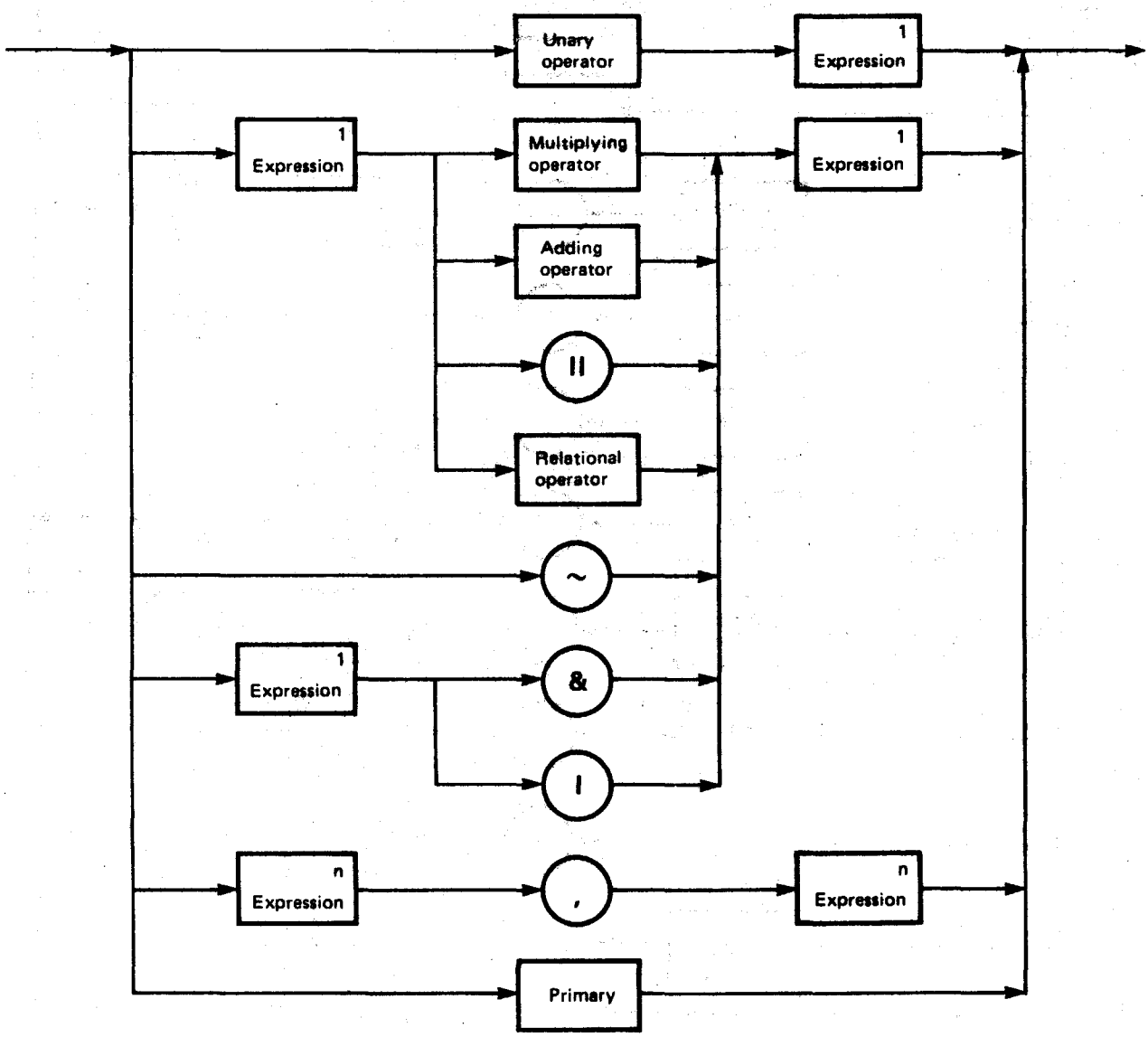
Field-Spec List



Tag-Spec List



Expression*

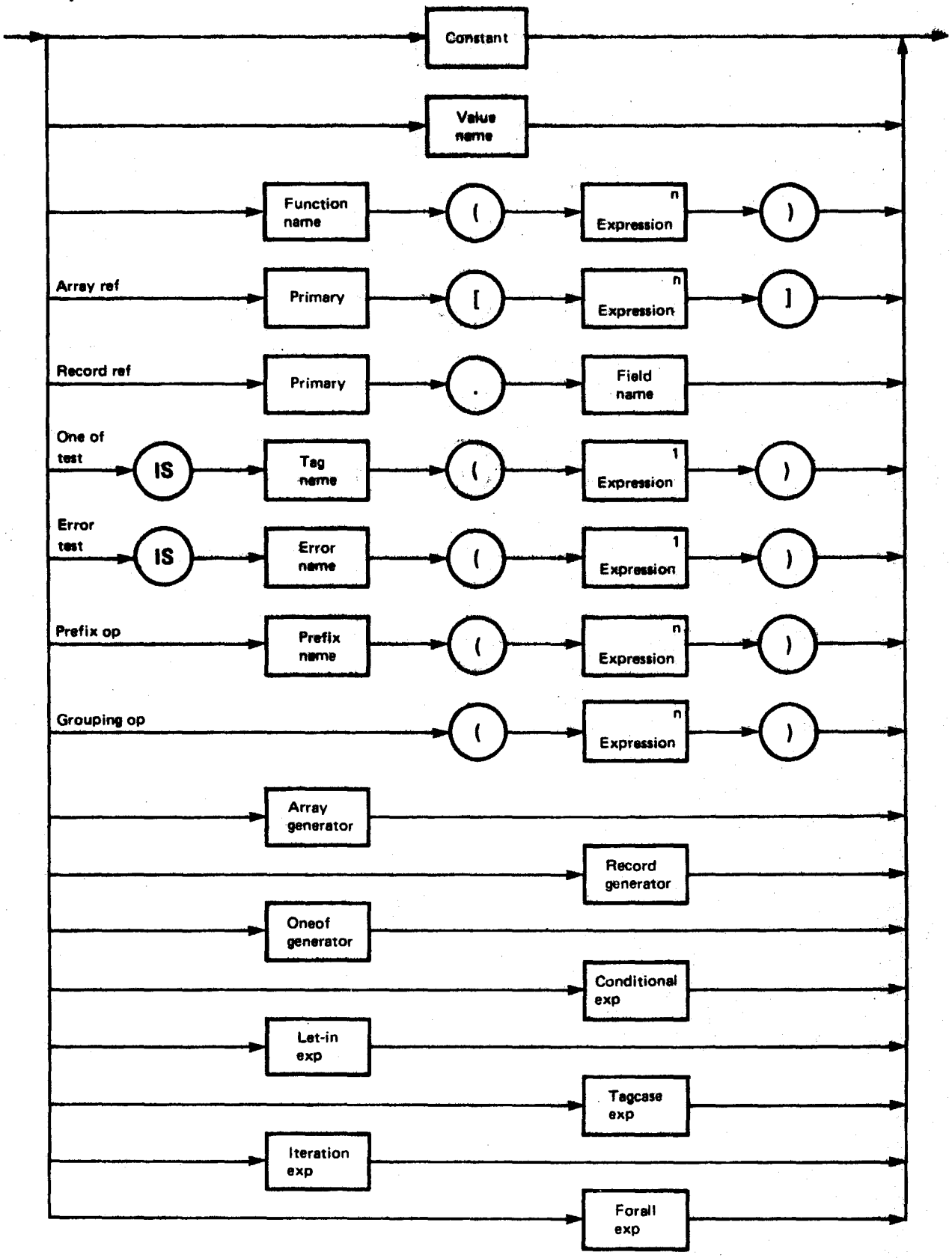


*The precedence levels for these infix operators is illustrated by their position in the chart; "unary operator" is highest precedence, comma lowest.

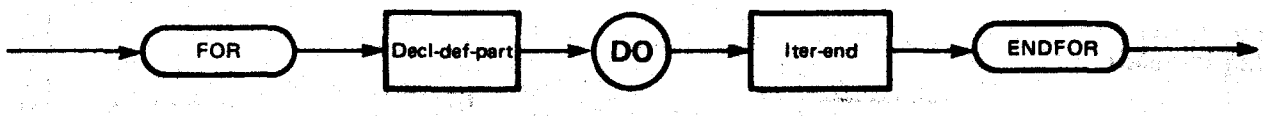
The superscript following "expression" indicates the number of values that must be represented by the term replacing that box in the program.

- an exact number → that arity is the only legal one
- "n" → any arity is valid
- "K" ⇒ arity must match arity of other expressions in some chart

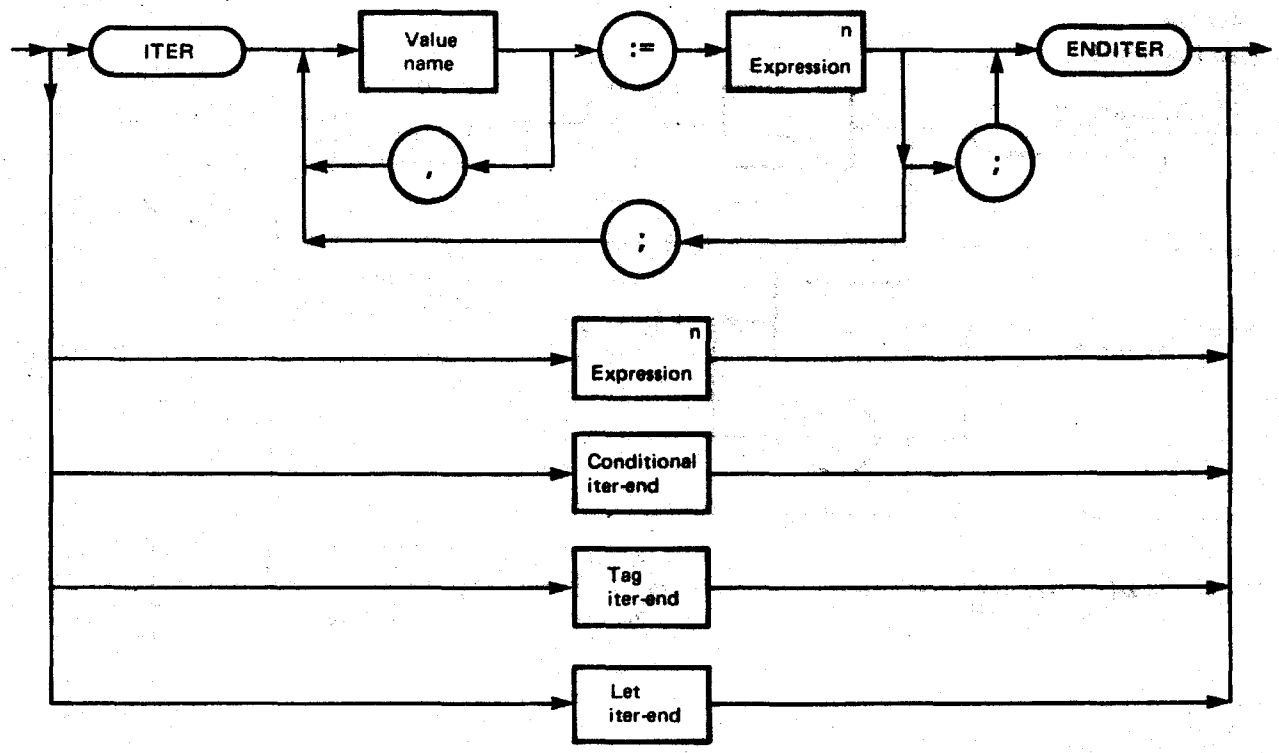
Primary


Constant
Value name
Function name
(
Expression n
)
Array ref
Primary
[
Expression n
]
Record ref
Primary
.
Field name
One of test
IS
Tag name
(
Expression 1
)
Error test
IS
Error name
(
Expression 1
)
Prefix op
Prefix name
(
Expression n
)
Grouping op
(
Expression n
)
Array generator
Record generator
Oneof generator
Conditional exp
Let-in exp
Tagcase exp
Iteration exp
Forall exp

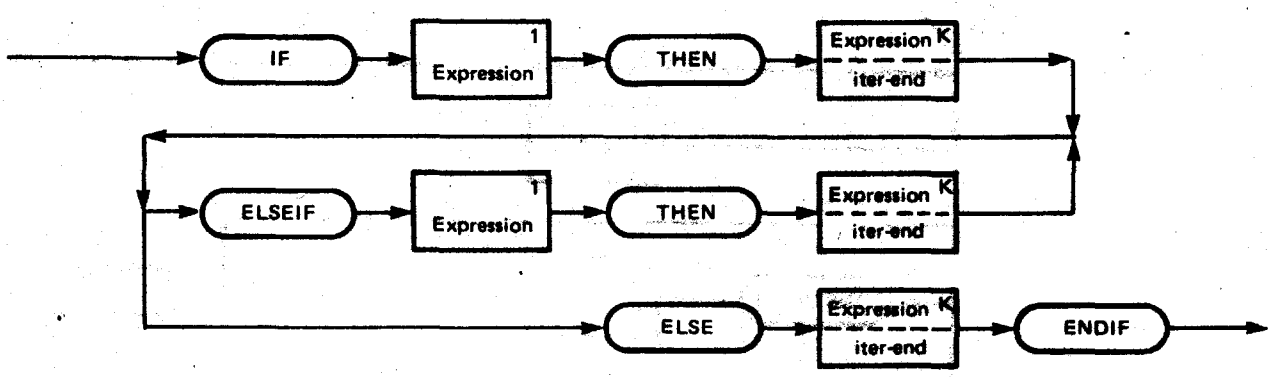
Iteration Exp



Iter-end

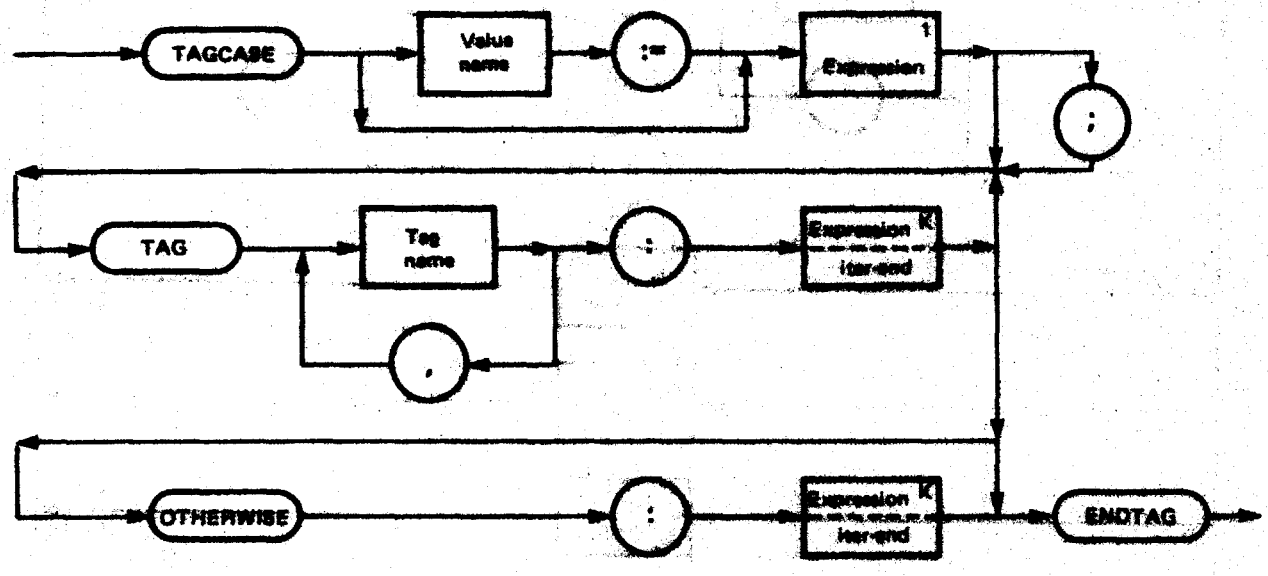


Conditional Exp
(conditional iter-end)

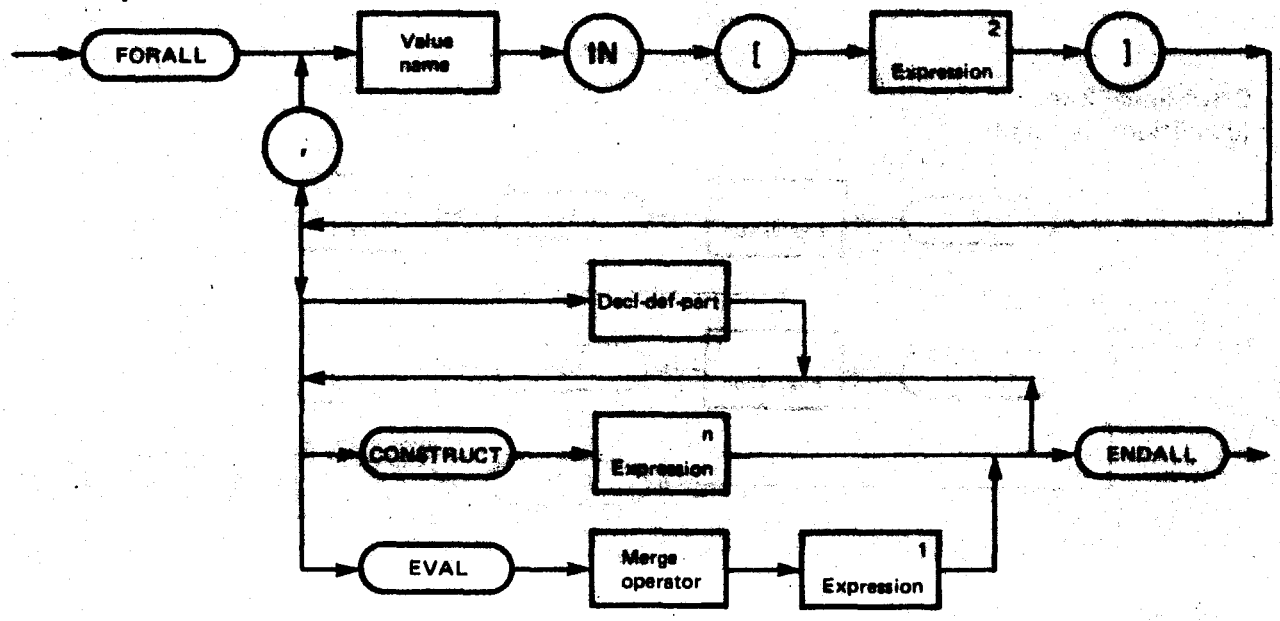


Let-in Exp
(let iter-end)

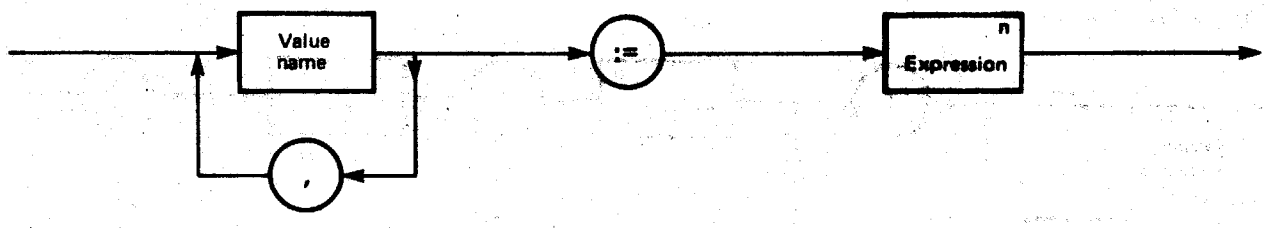
Tagcase
(tag iter-end)



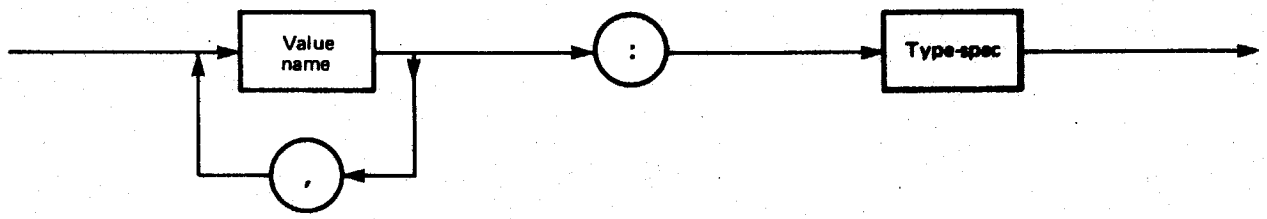
Forall Exp



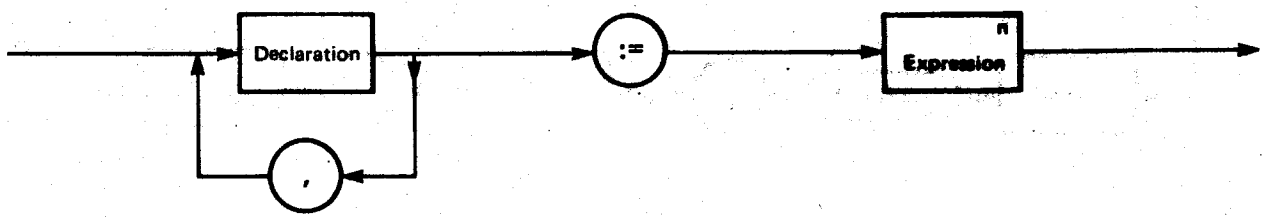
Definition



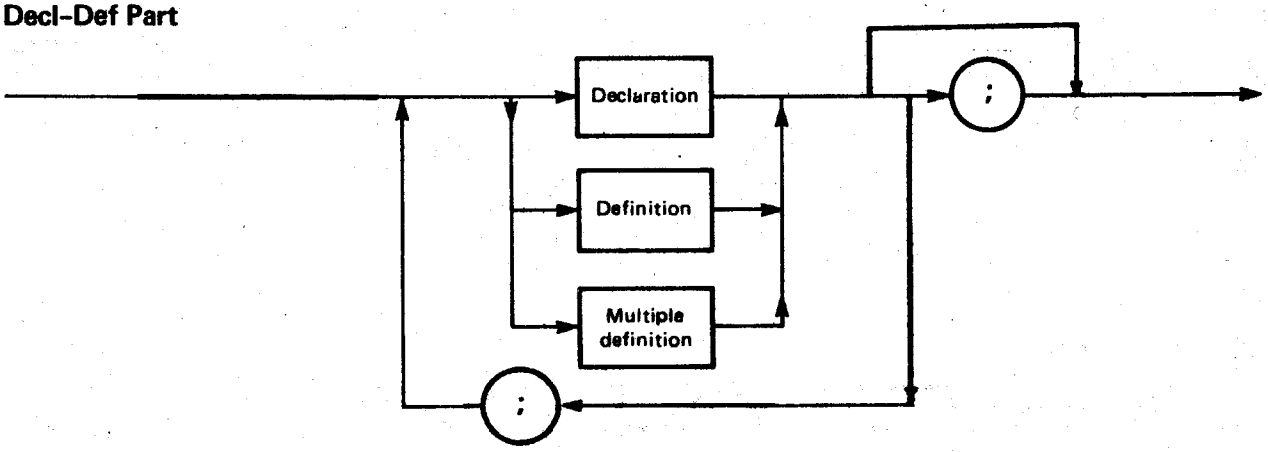
Declaration



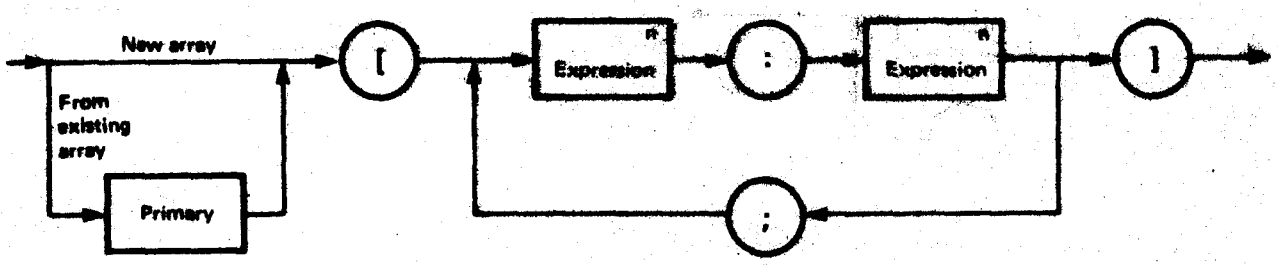
Multiple Definition



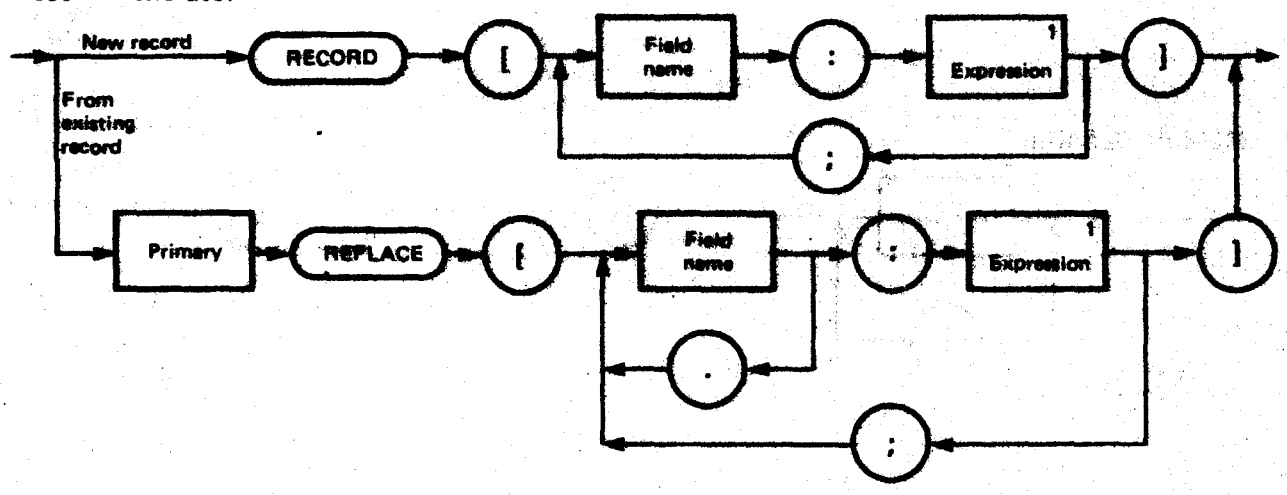
Decl-Def Part



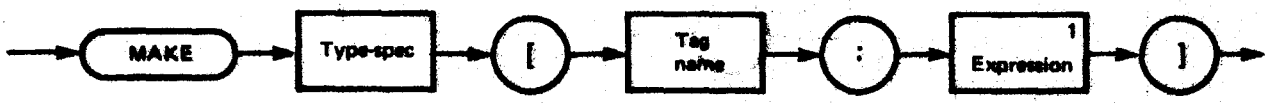
Array Generator



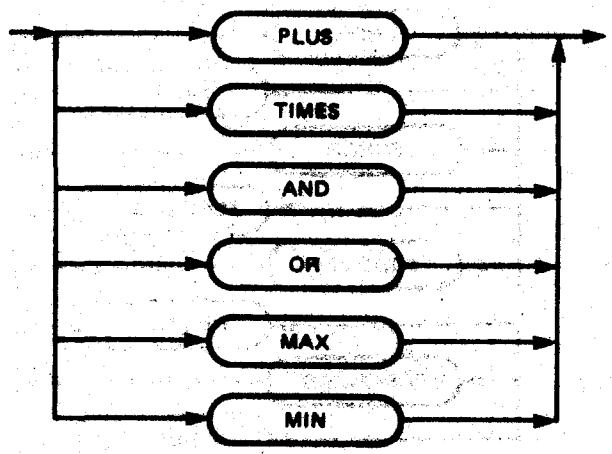
Record Generator



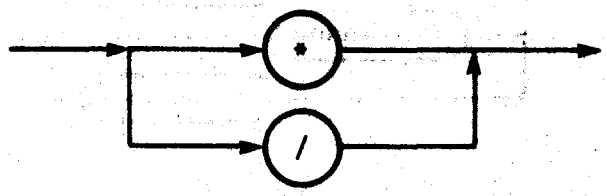
One of Generator



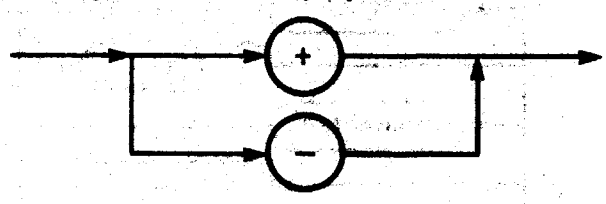
Merge Operator



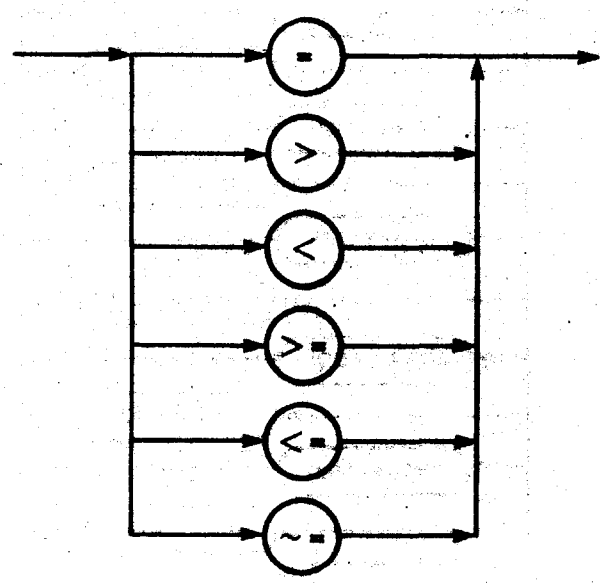
Multiplying Operator



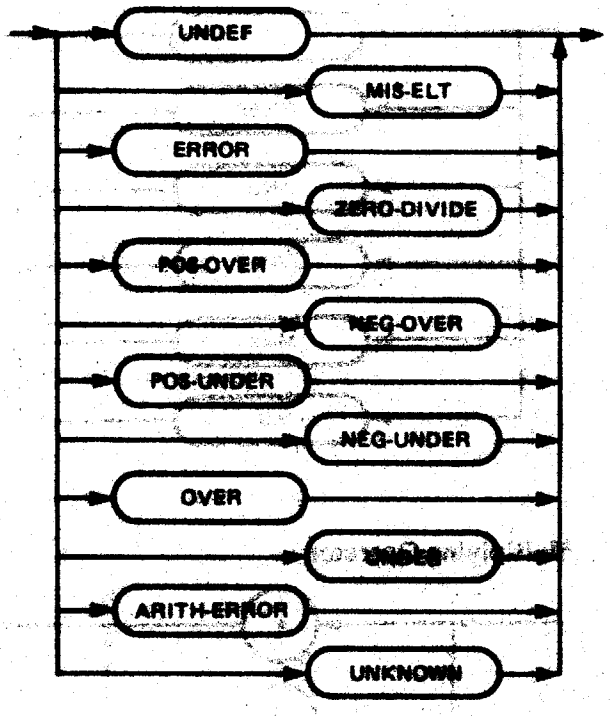
Adding Operator, Unary Operator



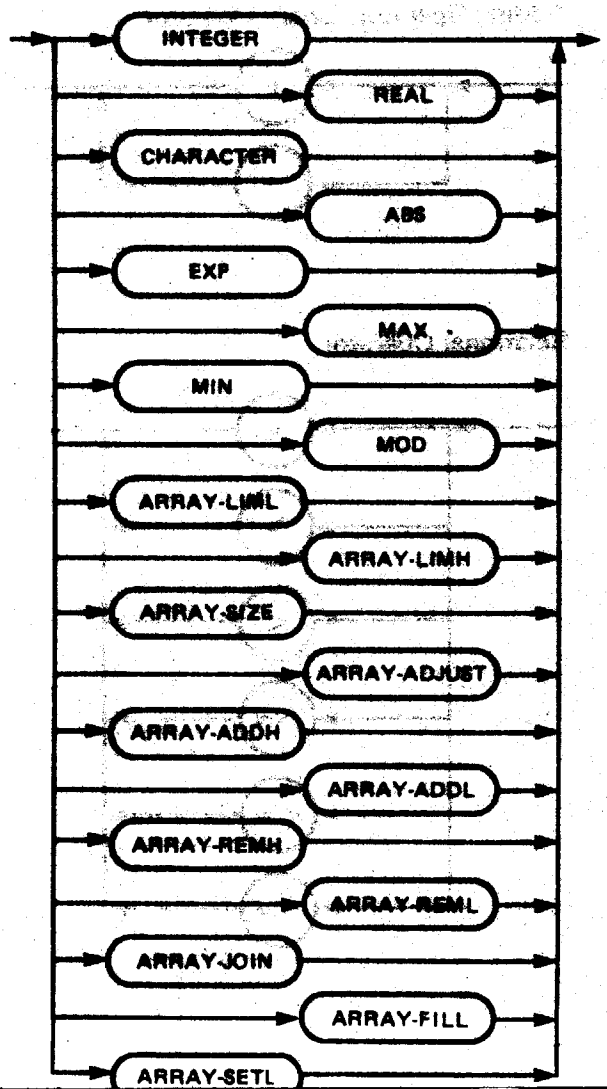
Relational Operator



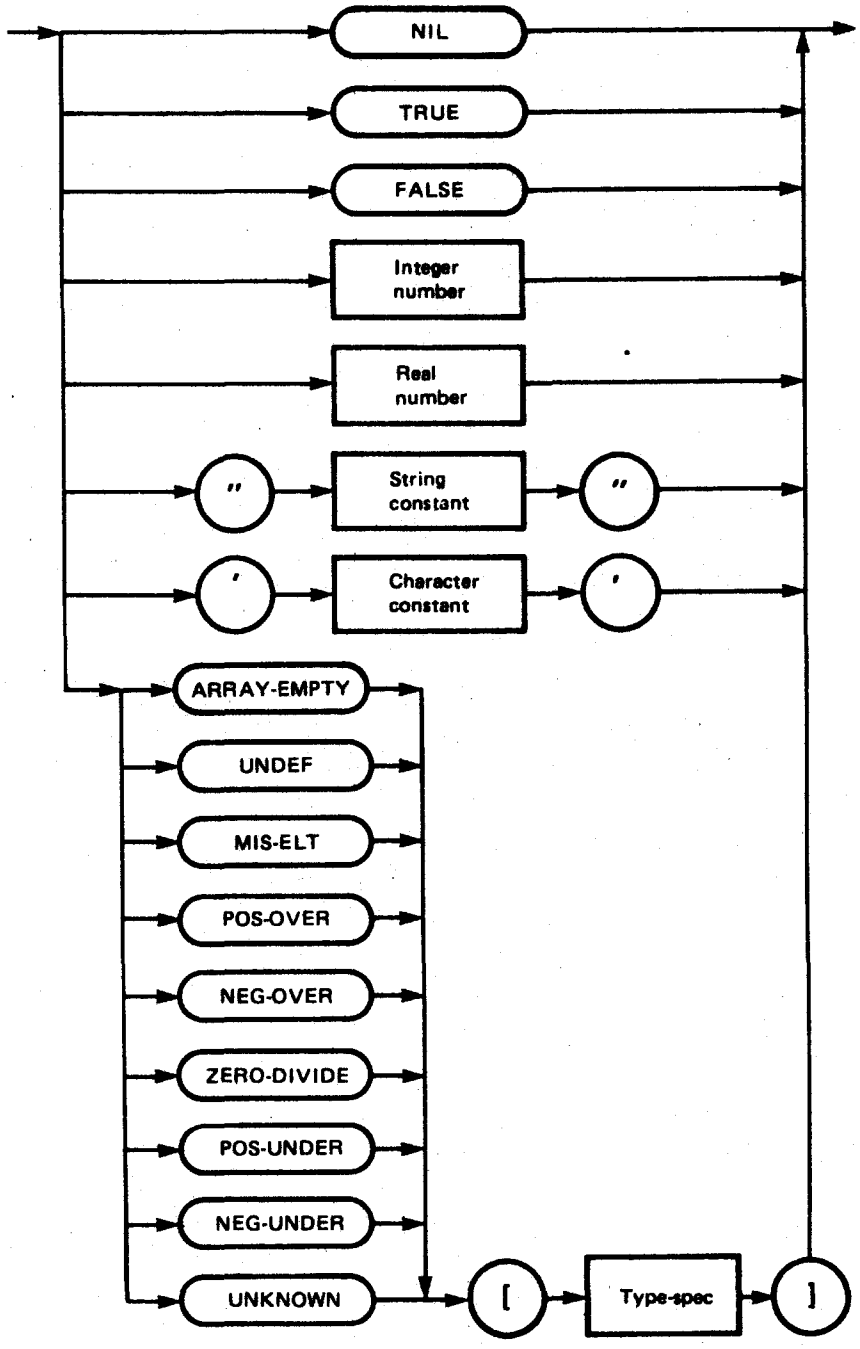
Error Name



Prefix Name



Constant



Function name, formal parameter, type name, value name, field name, tag name:

⇒ they are all simple identifiers