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. 8545103, 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 Brian 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_alt	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