The design and implementation of the programming language Natural

Alan L. Sparks

THE DESIGN AND IMPLEMENTATION OF THE PROGRAMMING LANGUAGE NATURAL

BY

ALAN LYLE SPARKS, 1960-

A THESIS

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

1984

Approved by

[illegible] (Advisor) John B. Prater

Paul D. Stigall

ABSTRACT

This paper reports progress on the development of the programming language Natural, currently under design by Dr. Thomas J. Sager at the University of Missouri-Rolla. Natural is a very high-level language with a mathematical flavor, and includes several concepts relatively uncommon in programming language design.

The text also discusses an implementation on the IBM Personal Computer of Mini-Natural, a subset of Natural, and presents examples of programs written in Mini-Natural.

ACKNOWLEDGEMENT

The author would like to extend his thanks to his advisor, Dr. Arlan R. DeKock, and committee members Dr. John B. Prater and Dr. Paul D. Stigall, for their invaluable assistance in compiling this text. Thanks also to Dr. Thomas J. Sager for allowing me a part in the Natural project. Special thanks to my family, John and Deborah, Star, Kevin, and all others who provided the support necessary to make it happen.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

I. INTRODUCTION

The focus of this chapter will be to examine features found in programming languages which are currently in use, noting features which accent or detract from the overall clarity and usefulness of programs written in these languages. The programming language Natural will then be discussed in light of these features.

A. REVIEW OF EXISTING LANGUAGES

Programming languages can be classified into three major divisions: conventional or procedural languages such as PL/I [1] and Pascal [2], dataflow languages such as VAL [3] and ID [4], and functional languages such as LISP [5] and FP [6]. Procedural languages are characterized by state-transition semantics, each state represented by a "statement." Procedural languages are common because the von Neumann machine uses state-transition semantics, and the translation of programs in these languages is made more direct and straightforward. Dataflow languages characterize programs as tree structures, where all operands to any operator can be evaluated concurrently. Concurrent processing provides an increase in execution speed over non-dataflow languages. Functional languages are characterized by a mathematical function notation and reduction semantics.

Programs written in functional languages are amenable to axiomatic analysis (useful to prove properties about programs) since their syntax closely resembles the mathematical notation utilized for expressing symbolic algebras.

The remainder of this section will compare languages in their treatment of eight important design topics. These topics are machine independence, storage representation, data structuring facilities, abstract operations, exceptions, variable scoping, the GOTO statement, and file input/output.

<u>Machine independence</u> is the principal feature which differentiates high-level languages from lower-level languages. The typical applications programmer is rarely concerned about the architecture of his machine, i.e. whether it has one accumulator or sixteen, allowable addressing modes, etc. By masking machine idiosyncrasies from the programmer, considerable detail is removed from the programmer's shoulder, and programs can more easily be ported to other machines.

The "memory" or <u>storage representation</u> of the von Neumann computer model is a collection of many bits, with little overlying structure. It is helpful to the programmer if a "type" can be associated with these storage elements, declaring a certain element to be an integer and another to be a Boolean variable, in order to impose some measure of structure on the store. A language with strong data typing associates with each identifier a type class (integer, Boolean, etc) and defines sets of operations which may be applied to data of these types. The compiler can ascertain the type compatibility of the operands presented to the language operators, thus reducing the possibility that the programmer will attempt to apply an operator to data of inappropriate type (adding Booleans, for instance).

<u>Data structuring facilities</u>, the means available with which to represent data, can play a major part in deciding the usability of the language for a particular application. PL/I has a multitude of data types and precisions, while APL [7] has only three. FORTRAN [8] has one data structuring facility (the array), while Pascal, with dynamic structures, is virtually unlimited. A minimal set of powerful, abstract data types is desirable, as too many types and precisions of data provide arbitrary options to the programmer and weak or insufficiently abstract types require the programmer to expend more effort designing needed data structures.

Many languages support the implementation of general dynamic data structures with pointers to areas of machine storage. Hoare [9] considers the introduction of the pointer into high-level languages as "a step backward from which we may never recover." Pointers create difficulties in program verification and debugging, forcing the user to deal with low-level details of memory allocation and reclamation. The clarity of programs which use pointers is questionable. Simulation of the linked-list data structure is the basis of most pointer applications; therefore, a high-level abstraction of the linked-list data structure would shift the responsibility of managing low-level machine details from the programmer to the language translator.

Operator abstraction, the ability to hide the data representation from the remainder of the program, is as valuable to the programmer as abstract data types. Lack of abstract operators has the effect of muddying the algorithmic specification of the program with details not intrinsic to the algorithm. Operator abstraction is accomplished in many languages by hiding details in subprograms. Much coding effort and programmer time can be saved, however, if the language operators are already "sufficiently" abstract.

Consider an algorithm for computing the sum of a collection of numbers. A suitable statement might be "set SUM to zero, then add each number x of sequence S into SUM." A typical coding of this algorithm in Pascal is

```
SUM := 0;
FOR I := 1 TO N DO SUM := SUM + S[I];
```

This construction implies much more than the original algorithm; namely, 1) an order in which to sum the elements, 2) a data representation (a vector), and 3) an access method for the individual elements. Better is the ability to write

```
SUM := 0;
FORALL X IN S DO SUM := SUM + X;
```

which is a more faithful expression of the original algorithm. Even better would be to remove all procedural specifications by writing

```
SUM := +/S
```

where the +/ operator specifies summation. FP and APL both offer such operations.

Exceptions, such as "overflow," "undefined," and "infinity," can be handled by a language in two manners. PL/I can detect and handle exceptional conditions with a programmer-written "ON-unit." Another means is to define special values to represent exceptions and define operators to return these values when exceptions occur. Either method will work, but the special values themselves often have uses not related to exception handling.

Infinity constants are useful when implementing algorithms which require the use of values known to be larger or smaller than any program-generated quantity (similar to MAXINT in Pascal). Finding the maximum or minimum value of a data set is an example of such an algorithm.

The "undefined" constant is useful to handle operations which, for some reason, have failed. Should an operation on the dataflow machine model fail, a mechanism by which the failing operator can terminate other pending operators must be implemented; a nontrivial task. Defining an error value for this situation and defining rules for propagating this value through operators eases implementation.

The occurrence of an evaluation error using this strategy can be detected by writing a test similar to

```
IF (X = UNDEF) THEN handle_error
```

The scope of a variable is the environment in which the value of that variable is accessible. Block-structured languages such as Pascal and PL/I extend the scope of a variable to all subprograms nested inside the block declaring the variable. Nested subprograms have both read and write access to nonlocal variables, and are free to modify these variables without using the standard parameter communication interface between calling and called routines. Doing so is termed a "side effect." The undesirability of side effects is well documented in the literature [2,10,11]. To reduce opportunities for side effects, either data transfer should be restricted to parameters and function return values, or nonlocal variables should not be write-accessible from an inner block.

A "patch" added to the scope rules of block-structured languages allows the global declaration of a variable to be superceded by a declaration in a local block. This serves as protection against side effects only if the declaration is not inadvertently omitted. Adding or removing a declaration can alter the semantics of the program totally.

Few languages are without the GOTO statement, admonished by Dijkstra [12] as being "too primitive" and "too much an invitation to make a mess of one's program." The unrestricted GOTO is a direct artifact of the von Neumann machine architecture and provides a control facility of little structure. The IF-THEN-ELSE and WHILE structures found in most newer languages removes much of the need for the GOTO statement. Knuth [13] proposed replacing the GOTO statement with a structured envelope he termed an "event indicator" which can be used to implement early block exits, the last remaining "good" use of GOTO.

The presence of GOTO statements also complicates optimization analysis of programs, since flowgraphs representing these programs cannot be guaranteed to be reducible (for a discussion of reducible flowgraphs, see Aho and Ullman [14]). BLISS [15] programs, with the GOTO statement replaced by a set of block-exit statements, are always expressible as reducible flowgraphs, considerably easing dataflow analysis and allowing optimization analysis to be performed concurrently with parsing.

File input and output is a controversial topic in language design. Wirth [16] proposed that files be viewed as a mathematical sequence with an identifiable upper bound, and included in the design of Pascal sequential files as an embodiment of the concept. FP does not define file operations because file access is inherently a nonfunctional operation (file writes cause side effects, while file reads should return a new value for each reference). Input-output is very difficult to axiomatize, making program verification by proof techniques currently impractical [10].

B. NATURAL

The programming language Natural [17] is presently under development at the University of Missouri-Rolla by Dr. Thomas J. Sager. The principal aim of the design project is to provide "a vehicle for expressing abstract programming concepts clearly and precisely in a natural and mathematical form." [18]

To realize this goal, three design criteria were outlined when the project was initiated. The aim of the Natural project was to:

1. Integrate the strong points of functional languages such as LISP and FP, and conventional

languages such as Pascal and PL/I;

2. Promote structured programming techniques by providing versatile and abstract selection and iteration facilities and data structures;
3. Provide for explicit and implicit expressions of parallelism.

Implicitly present among these three points are the goals of simplicity and uniformity.

Natural, like APL and FP, has a decidedly mathematical syntax. It is hoped that "the succinctness and descriptive power of a mathematicized language will enable the user to depict complex processes in their totality, in decisive detail, and in a form free of abstractly irrelevent detail [19]."

Natural is designed to be largely machine-independent. The primitive operations (sequence concatenation, set inclusion, etc.) have no direct hardware equivalent in conventional von Neumann computer models. Features of other languages which mirror hardware features have either been abstracted (the sequence structure) or eliminated (the GOTO statement).

The structure of Natural is a combination of features found in the three languages classes discussed in the previous section. The language is similar in nature to procedural languages, since they are familiar and simpler to translate for existing architectures.

Algorithms are constructed by defining functions, as in functional languages, permitting employment of proof techniques designed for functional languages on Natural programs. Constructs for expressing parallelism have been included, giving Natural some of the power of dataflow languages.

Natural defines one precision of integer, real, character, and Boolean scalar data. Type compatibility is strictly enforced by the compiler. Conversions from one data type to any other type is possible, but must be explicitly specified by the programmer by referencing the predefined type-coersion functions. Three structured types, functions, sets, and sequences, are defined in the language. Structured types are homogeneous collections of scalar or structured values. Functions are code bodies which are evaluated on some set of parameters to yield an output value. Sets are an embodiment of the set concept in mathematics. Sequences are ordered lists of randomly-accessible values.

The structured types can define data in more than one way. This has three advantages. The first is flexibility; for example, sets can be defined either by enumerating their elements, as in Pascal, or by a code body which represents the set's characteristic function. Secondly, the language can include statements which allow abstract specification of operations, such as the FORALL statement, which parallels the FOR statement of Alphard [20]. A third advantage is that a suitably smart compiler can analyze the abstract source program and choose an appropriate representation automatically [21]. If, for instance, a programmer coded a difficult-to-compute function with a limited domain, the compiler might choose, in order to optimize execution speed, to precompute each value within the function's range, and store these values in an array.

Infinity is represented in Natural by the constants NEGINF and POSINF, representing negative and positive infinity, respectively. The constant UNDEF represents the quality "undefined." The constants are typeless, and may be assigned to or compared against any type of variable.

The scope rules of Natural are similar to those found in block-structured languages such as PL/I and Pascal, i.e. a nested function has access to the variables declared in its enclosing block. The values of these nonlocal variables are not alterable by the nested function, however - copies of the nonlocal variables are manipulated within the nested definition (analogous to call-by-value). Natural binds the nonlocal reference to the value the nonlocal variable contained at the time of function definition. The objective of this strategy is to have all functions fully defined when declared, rather than leaving "holes" which reduces the function's independence from other program modules.

To clarify, consider Figure 1. Natural binds the nonlocal reference to the value of A when C is defined; hence the value 10 is assigned to B.

The GOTO statement is conspicuously absent from Natural. It is replaced by the PENDING statement, which is a syntactic variation of Knuth's event indicator. This statement, together with the decision and iteration statements, presents a minimal but complete set of operators.

Files are conceptually similar to sequences, in that a file is an ordered sequence of elements with a known upper bound or size. Input to and output from Natural programs is accomplished by binding files to variables declared as sequences of the appropriate element type, and using the sequence operators to read and write the file. Therefore, once axioms describing the semantics of sequence operations are invented, they can be applied directly to the semantics of file I/O.

```
Let
     a be int <- 5;
     b be int;
     c be func (int -> int) <- [output <- input * a]
Do [
     a <- 10;
     b <- c (2)
   ]
End
```

Figure 1. Example of nonlocal reference

II. THE MINI-NATURAL LANGUAGE SUBSET

This chapter is intended as a description of the Mini-Natural language subset implemented during this research. The reader is referred to Appendix A for a summary of Mini-Natural syntax.

A. <u>OMISSIONS FROM AND RESTRICTIONS ON NATURAL</u>

The full Natural language is described by Sager [17]. To reduce the project to a size which could be completed and tested in the time available, certain features were restricted or omitted entirely from the Mini-Natural definition:

1. Parallel processing features were not included.
2. Input/output is defined only for p-System diskette files. Device I/O is not implemented.
3. Concatenation and selection of a single element are the only sequence operations.
4. The real data type is omitted. Sequences and sets may contain only scalar elements, and functions can accept and return only scalar values.
3. The FOR and PENDING control structures are not included in the Mini-Natural definition.

Many features present in Natural have not been discussed, and have been omitted from the Mini-Natural language definition.

B. LEXICAL CONVENTIONS

A Mini-Natural source program is composed of constants, identifiers, and operators (collectively termed "tokens"). Blanks, tabs, and carriage returns (termed "separators") may be inserted between tokens as the user wishes to separate adjacent tokens and enhance readability. Separators may be omitted if doing so results in no ambiguity in the specification of identifiers and constants.

The sequence

```
>> any sequence of characters
```

may be included in the source text anywhere a separator is allowed, without affecting the semantics of the program. The compiler ignores any commentary between the ">>" and the next carriage return.

1. Constants: Mini-Natural defines three classes of constants: integer, character, and Boolean. Integer constants are decimal numbers, optionally preceded by a sign. Mini-Natural restricts integer constants to the range -32768 to +32767. Character constants are written 'x, where x is any printable ASCII character. Note there is no closing quote, as in other languages.

The remaining constants are summarized below:

Identifier	Type	Semantics
true	Boolean	Logical truth
false	Boolean	Logical falsity
eoline	char	End-of-line character
undef	typeless	Undefined
neginf	typeless	Negative infinity
posinf	typeless	Positive infinity

2. Identifiers and Keywords: An identifier consists of a letter, optionally suffixed by one or more letters or digits. Identifiers may be of any length, but only the first eight characters are significant. Uppercase and lowercase differences between letters are ignored.

The following identifiers are reserved as keywords in Mini-Natural and may not be used as user-defined names:

be	end	input	posinf	text
bool	eoline	int	pred	true
char	external	let	pwrseq	type
choose	false	mod	pwrset	ubnd
div	func	neginf	readbool	undef
do	if	next	readint	while
else	in	output	succ	

C. <u>PROGRAM STRUCTURE</u>

All Mini-Natural programs consist of a "LET statement" followed by the keyword <u>end</u> to signal the logical end of the source program.

```
program:
     letstmt end

letstmt:
     let declarations do simplestmt

declarations:
     declaration { ; declaration }
```

D. <u>DATA TYPES AND DECLARATIONS</u>

All identifiers must be declared before use. Declarations serve to associate data types with identifiers.

```
declaration:
     identifier { , identifier } be type
```

Data types are grouped into two major divisions: scalar types and structured types. Structured types are compositions of scalar types, and can be further separated into three classes: sets, sequences, and functions.

1. <u>Scalar Types</u>: The allowable scalar types are <u>int</u>, <u>char</u>, and <u>bool</u>, for integer, character, and Boolean data, respectively. Each scalar variable holds a single data value.

2. Function Type: Function variables are declared

type:
    func (intype -> outtype)

where intype and outtype are scalar types. The function accepts a scalar parameter of type intype and returns a scalar of type outtype.

Functions are referenced by coding the expression f(x), where f is a defined function name and x is an expression of f's intype. Within a function body, the argument passed to the function is accessible through the identifier input. The output of the function can be defined by assigning a value to the identifier output. If no assignment to output occurs within the function, the value of the function is undef.

3. Set Type: Set variables are declared

type:
    pwrset (basetype)

where basetype is any scalar type. The set consists of scalars of the specified base type. Mini-Natural treats sets as though defined func (basetype -> bool).

A set may be defined either by enumerating its elements or by associating with it a statement which represents the "characteristic function" of the set. Enumerations are written

set-enumeration:
    { expr { , expr } }

where expr is a scalar expression of the specified base type. neginf, posinf, and undef are not allowable set elements. Sets of int are restricted to the integers 0 through 4079. The { } brackets are called "set builders."

4. Sequence Type: Sequence variables are declared

```
type:
    pwrseq (basetype)
```

where basetype is any scalar type. Sequences are vectors of scalars of the specified base type. Mini-Natural treats sequences as if defined func (int -> basetype).

A sequence can be defined either by enumerating its elements or by associating with it a statement which specifies the values included. Enumerations are written

```
sequence-enumeration:
    <. expr { , expr } .>
```

where expr is a scalar expression of the specified base type. The <. .> brackets are called "sequence builders."

A shorthand notation is available for describing a sequence of characters. The text string "STUFF" is equivalent to the sequence <. 'S, 'T, 'U, 'F, 'F .>, but is clearly easier to write. To include a quotation mark inside a text string, simply code two adjacent quotation marks.

5. Initialization: Variables are automatically initialized to undef when declared. This may be overridden, and a different initial value specified, by

appending an assignment to the declaration. Only one variable may be initialized per declaration.

```
     declaration:
          identifier be type <- expr
          identifier be type <- stmt
```

Expr is an expression, which is evaluated. Stmt is a statement. The type of expr must be consistent with the type of identifier. Statements may be assigned to any structured type.

Within the LET statement which defines the program, sequences may have an initialization of the form

```
     declaration:
          identifier be type <- external name
```

where type is a sequence definition and name is a character sequence enumeration or text string containing the name of a p-System diskette file. The sequence is bound to the current contents of the external file (the empty sequence if the file did not previously exist). The final value of the sequence is saved on the external medium upon normal program termination.

6. <u>Type Declarations</u>: Type names may be declared in Mini-Natural by a declaration of the form

```
     declaration:
          identifier be type type
```

An identifier so declared may be used as a synonym for type in subsequent declarations.

E. EXPRESSIONS

In the following discussion, operators with a precedence level of 1 are executed before operators with precedence 2, which are executed before precedence 3 operators, etc. Parentheses can be used to alter the order of evaluation of an expression.

1. Scalar Operators: Table I lists the available scalar operators.

If an operand of any int operator has the value posinf, neginf, or undef, the operation yields the value undef. Also, if the second operand to the division or mod operator is zero, the operation yields undef. Table II enumerates the values of the bool operators for true, false, and undef. posinf and neginf act identically to undef.

2. Relational Operators: The available relational operators are listed in Table III. All relational operators have a precedence level of 2.

"Comparability" tests whether two elements have attributes in common. A comparability test against any two scalars, sequences, or functions yields true. A comparability test against two sets yields true if the sets are not disjoint. undef tests as comparable only to undef.

For the remaining relational operators, neginf is

TABLE I

SCALAR OPERATORS

OPERATION	TYPE OF OPERAND(S)	TYPE OF RESULT	PRECEDENCE	MEANING	NOTES
+ x	int	int	1	Identity	1
- x	int	int	1	Negation	1
x + y	int	int	5	Addition	1
x - y	int	int	5	Subtraction	1
x * y	int	int	4	Multiplication	1
x / y	int	int	4	Integer division	1,2
x mod y	int	int	4	Modulo division	1,2
^ x	bool	bool	1	Not	1
x & y	bool	bool	4	And	4
x | y	bool	bool	5	Or	4
x @ y	bool	bool	5	Exclusive OR	4
x \ y	bool	bool	5	And Not	4
x div y	int	bool	2	x evenly divides y	1,3
x ^div y	int	bool	2	^(x div y)	1,3

NOTES: 1. Yields undef for x and/or y undef, posinf, neginf.

2. Yields undef if y is zero.

3. Yields undef if x is $\leq$ zero.

4. See Table II.

TABLE II

VALUE OF BOOLEAN OPERATORS

X	Y	X & Y	X | Y	X \ Y	X @ Y
false	false	false	false	false	false
false	true	false	true	false	true
true	false	false	true	true	true
true	true	true	true	false	false
undef	undef	undef	undef	undef	undef
undef	false	false	undef	undef	undef
undef	true	undef	true	false	undef
false	undef	false	undef	false	undef
true	undef	undef	true	undef	undef

TABLE III

RELATIONAL OPERATORS

OPERATOR	SCALARS (Note 1)	SEQUENCES (Note 2)	FUNCTIONS	SETS
X = Y X ^<> Y	$X = Y$	$X = Y$	$X = Y$	$X = Y$
X <> Y X ^= Y	$X \neq Y$	$X \neq Y$	$X \neq Y$	$X \neq Y$
X < Y X ^>= Y	$X < Y$	$X < Y$	illegal	$X \subset Y$
X <= Y X ^> Y	$X \leq Y$	$X \leq Y$	illegal	$X \subseteq Y$
X > Y X ^<= Y	$X > Y$	$X > Y$	illegal	$X \supset Y$
X >= Y X ^< Y	$X \geq Y$	$X \geq Y$	illegal	$X \supseteq Y$
X <=> Y	X is "comparable" to Y (see text)			
X ^<=> Y	Equivalent to ^(X <=> Y)			

NOTES: 1. Linear ordering, e.g. 1 < 2 and <u>false</u> < <u>true</u>

2. Lexicographic ordering, e.g.
<.1, 2.> < <.1, 3.> and
<.'A, 'B.> < <.'A, 'B, 'C.>

less than any value (except itself, to which it is equal), and posinf is greater than any value (again, except itself, to which it is equal). undef is equal only to itself; other comparisons involving undef yields the value undef.

Functions can be compared only for equality, non-equality, or comparability. Functions are equal if they are aliases.

3. Set Operators: The available set operations are listed in Table IV.

The in operator tests for set membership. If s is a set of basetype t, and x is a scalar of type t, the expression x in s returns true if x is a member of s, false otherwise. x ^in s is equivalent to ^(x in s). The in operator has a precedence level of 2.

4. Sequence Operators: The concatenation operator || takes two sequences of common base type as operands and produces a new sequence, the second sequence "hooked onto" the end of the first sequence. The concatenation operator has precedence 4.

If s is a sequence, s`ubnd and s`next are variables of type int. When used in an expression, s`ubnd returns the current length of s. In the same context, s`next returns the index of the element following the last element referenced in s. next can be used to iterate

TABLE IV

SET OPERATORS

OPERATOR	PRECEDENCE	MEANING
^ S	1	Complement of set S, e.g. the universal set U - S
S | T	5	Union of sets S and T
S & T	4	Intersection of sets S and T
S \ T	5	Relative difference of sets S and T, e.g. the set S with members of set T removed
S @ T	5	Symmetric difference of sets S and T, e.g. (S-T) | (T-S)
x <u>in</u> S	2	Membership of x in set S
x <u>^in</u> S	2	Equivalent to ^ (x <u>in</u> S)

through a sequence.

The assignment s`ubnd <- expr changes the upper bound of s to the value of expr. The sequence is truncated or extended with undef values on the right. s`next <- expr sets the next index to the value of expr. In either case, a runtime error occurs if expr evaluates to undef, posinf, or neginf.

If s is a sequence and n a scalar expression of type int, s(n) returns the nth element of s. If n is less than 1 or greater than the number of elements in s, or if s or n has the value undef, posinf, or neginf, s(n) returns undef. s`next is automatically set to n+1.

s(n) <- expr, where n and expr are scalar expressions of type int, assigns the value of expr to the nth element of s. A runtime error occurs if n is less than 1. s`next is automatically set to n+1.

F. EXECUTABLE STATEMENTS

The action of a program or function (the "DO" part) is specified by a statement.

```
stmt:
     letstmt
     ifstmt
     whilestmt
     compoundstmt

simplestmt:
     ifstmt
     whilestmt
     assignstmt
     compoundstmt
```

1. <u>LET Statement</u>: The LET statement permits the declaration of one or more variables and a new "scope."

```
letstmt:
     let declarations do simplestmt
```

Declarations were introduced in section D. The variables declared are automatically created and initialized before executing simplestmt.

2. <u>Assignment Statements</u>: The assignment operator <- is used to define new values for variables.

```
assignstmt:
     variable <- expr
     variable <- stmt
```

If the first form is coded, expr is an expression, which is evaluated. The value of the expression must be compatible with variable. If the second form is used, variable is defined as a function with definition stmt. Statements may not be assigned to scalar types.

3. <u>IF Statement</u>: The IF statement allows conditional execution of one or more statements.

```
ifstmt:
     if ( clause {; clause } [ else simplestmt ] )

clause:
     expr -> simplestmt
```

expr is an expression which must evaluate to a scalar Boolean value. The expression portion of each clause is evaluated in turn, left to right. The simplestmt part of the first clause whose expr part evaluates to <u>true</u> is executed, and the IF structure terminates. If no expression evaluates to true, the else statement is executed if present and the IF structure terminates.

4. WHILE Statement: The WHILE statement is used to cause repetitive execution of one or more statements.

```
whilestmt:
     while ( clause { ; clause } )
clause:
     expr -> simplestmt
```

expr is an expression which must evaluate to a scalar Boolean value. The simplestmt portion of the first clause is repeatedly executed while the expr portion of the first clause is true; then the simplestmt portion of the second clause is repeated while the expr portion of the second clause is true; and so on. The structure returns to evaluate the first clause after the last clause terminates. The WHILE structure terminates when the expression part of no clause evaluates to the value true.

5. Compound Statement: The IF and WHILE clauses allow only one action to be specified per clause. A sequence of actions can be specified with the compound statement.

```
compoundstmt:
     [ simplestmt { ; simplestmt } ]
```

The [ ] bracket pair serve to group multiple statements into a single unit. Statements in a compound statement are executed left to right.

G. BUILT-IN FUNCTIONS

Built-in functions are referenced from Mini-Natural by coding an expression with the syntax:

```
bifreference:
     bifname ( expr )

bifname:
     bool           int           readint
     char           pred          succ
     choose         readbool      text
```

where expr is an expression whose type depends on which built-in function is named.

1. Type Conversion - INT, CHAR, BOOL, and TEXT:

These functions convert a scalar argument of any type to the type indicated by the function name.

int (x) returns an integer corresponding to the internal coding of x (true=1, false=0, ASCII collating sequence). If x evaluates to posinf, neginf, or undef, int returns undef.

char (x) converts x to an integer and returns the x'th character in the ASCII collating sequence. If x cannot be converted to an integer in the range 0..255, char returns undef.

bool (x) converts x to an integer and returns true for x=1, false for x=0. If x cannot be converted to either 0 or 1, bool returns undef.

text (x) converts x into a character sequence. The result is a character representation of the value of x, readable by the *readbool* and *readint* functions. *true* is converted to the sequence <. 'T, 'r, 'u, 'e .>, *false* to <. 'F, 'a, 'l, 's, 'e .> . If x evaluates to *posinf*, *neginf*, or *undef*, *text* returns *undef*.

2. *Data Conversion - READINT and READBOOL*: These functions accept a character sequence as argument. Both functions begin extracting characters from the sequence x at x‘*next*. Leading blanks are discarded, and scanning stops with the first trailing blank or comma, or with the end of the sequence. x‘*next* will index the first unprocessed character in x (if any).

readint (x) reads a sequence of characters from x and interprets the sequence as an optionally signed integer. If the characters do not form a valid integer constant, *readint* returns *undef*.

readbool (x) scans x for the first character not a comma or blank. If the character found is a T, *readbool* returns *true*; if the character is an F, *readbool* returns *false*. Characters following this character to the next blank or comma or the end of the sequence are discarded. If neither T or F is found, *readbool* returns *undef*.

3. *PRED and SUCC*: These functions accept a scalar argument of any type. *pred* (x) returns the immediate predecessor of x (or *undef* if no predecessor exists),

while _succ_ (x) returns the immediate successor of x (or _undef_ if no successor exists). Both functions return _undef_ if the argument evaluates to _undef_, _posinf_, or _neginf_.

4. _CHOOSE_: _choose_ (x) returns an arbitrary element extracted from x, where x is a set of any type. If no elements exist, _choose_ returns _undef_. The current implementation also returns _undef_ if x is represented by a statement.

III. IMPLEMENTATION OF THE MINI-NATURAL SUBSET

A. OVERVIEW

Mini-Natural was implemented in Pascal on the IBM Personal Computer, running the UCSD p-System operating system. The system consists of two major components: the Compiler, and the runtime Interpreter.

Natural is compiled to an intermediate language, composed of fixed-format quadruples, and interpreted. This scheme was chosen because the internal representation of data contained within structured variables can change during execution, and because the operators of Natural have few direct equivalents on the machine level. Full compilation would entail either a high percentage of runtime system calls, or inline operator simulation, which would greatly increase the size of the compiled code. Note that SNOBOL uses the same technique for similar reasons [21].

B. THE INTERMEDIATE CODE

The Compiler generates quadruples containing an operator, two operand descriptors, and a destination descriptor. These "quads" represent the machine language of the hypothetical Natural machine, a machine containing 500 elements of storage and five registers, a Program

Counter (PC), a Stack Pointer (TOP), a local data pointer (LCL), a global data pointer (GBL), and an auxiliary (AUX) register.

Appendix B lists the intermediate code operators. The operator field has an optional mask field, used by the Interpreter to select variations of certain operations. Operand descriptors, illustrated in figure 4, contain information regarding the addressing mode of the operand, as well as operand type information.

C. THE COMPILER

Mini-Natural is compiled to intermediate code using a one-pass compiler. Forward references are backpatched when resolved. The Compiler is implemented in three separately-compiled units: the Lexical Analyzer, the Syntax/Semantic Analyzer, and the Code Generator.

1. The Lexical Analyzer returns one token per call by the Syntax/Semantic Analyzer, and uses ad-hoc methods to recognize and classify tokens from the program source file. Also contained in the Lexical Analyzer module are source-handling facilities for reading source characters, generating the source listing, and writing error messages.

2. The Syntax/Semantic Analyzer uses recursive-descent parsing techniques to analyze tokens retrieved from the Lexical Analyzer. Code is generated by calling the Code Generator modules with operator and operand descriptors.
3. The Code Generator provides facilities for storing generated quadruples into a code array, and for dumping the code array to external storage when compilation is complete. The Code Generator also provides procedures which generate the program and subprogram prologue and epilogue code.

Providing error recovery is beyond the scope of this research. When an error is detected, a pointer to the current token and an error message is displayed, and compilation is halted.

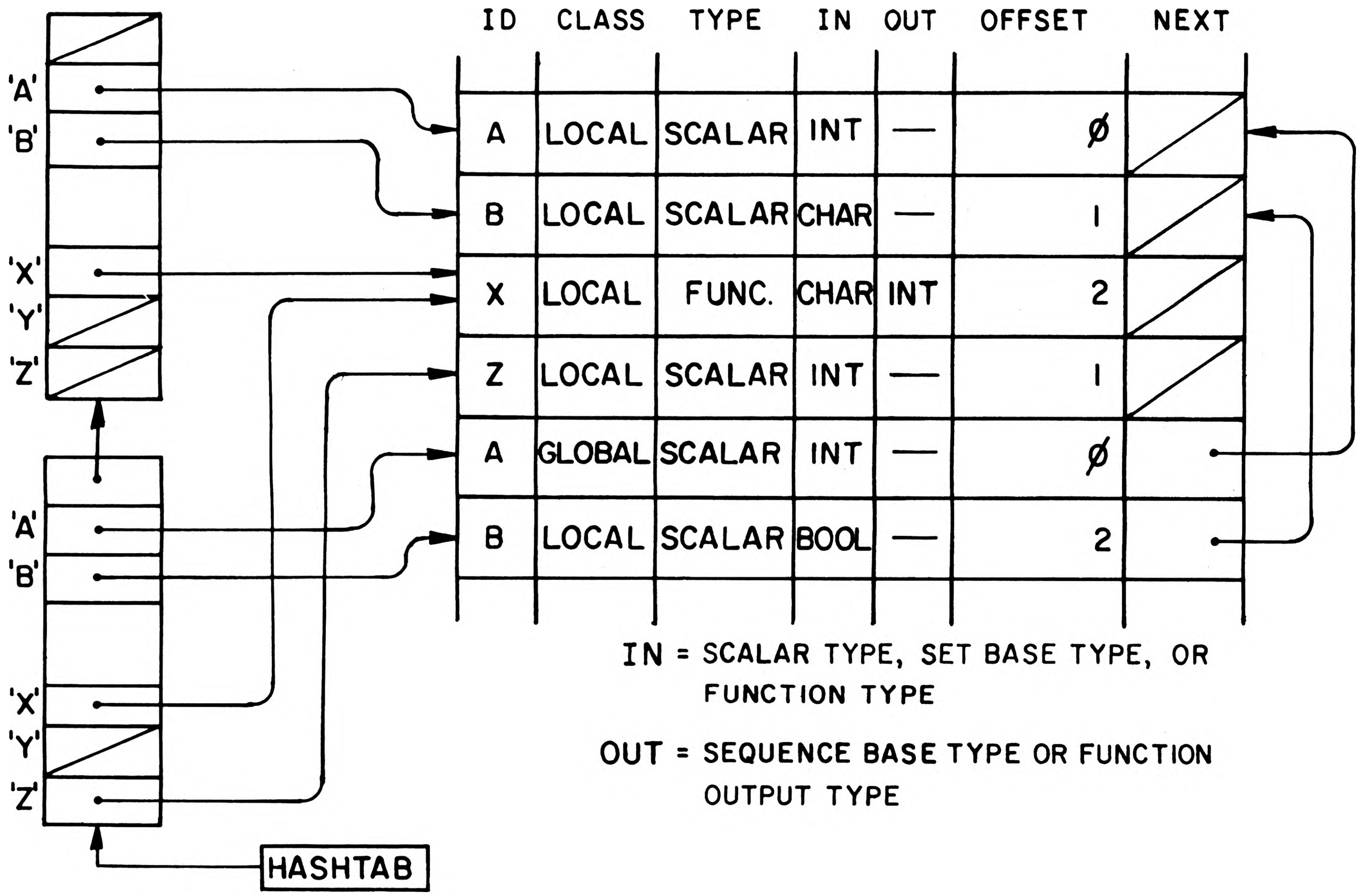
A stack symbol table, similar to that described by Calingaert [22], is utilized to maintain information pertaining to declared variables. A hash table, with one pointer corresponding to each letter of the alphabet, points to a chain of symbol table entries for identifiers beginning with that letter. Newly-declared identifiers are entered into the symbol table by adding an entry to the top of the symbol table and splicing the entry onto the front of the corresponding hash chain. This guarantees that a forward search of the chain will find the identifier occurrence declared in the most-recently-opened scope first. Scopes are opened by creating a copy of the hash table for the enclosing scope, and closed by discarding the hash table for that scope (freeing symbol table storage). Figure 3 shows a view of the symbol table mechanism while compiling the program fragment of figure 2.

Storage for declared variables is allocated from stack space. The Compiler assigns to each variable an offset from the LCL register, and uses indexed addressing for storage references. Offset assignments begin at LCL offset 1; offset 0 is reserved for the variable _output_. _Input_ is allocated at LCL offset -4, the top-of-stack before variable storage allocation. Offset assignments begin at 0 for the main program, which has no _input_ or _output_.

Nonlocal variable references are handled by leaving space at the end of the function body for an operand descriptor, and generating a quadruple to copy the current value of the non-local variable into the operand descriptor. References are indexed by the GBL register, which is initialized by the subprogram prologue code to point to the first element of the global pool.

```
Let
    A be int  <- 6;
    B be char;
    X be func (char -> int)  <-
       Let
           Z be int <- A;      >> A is nonlocal
           B be bool
       Do  . . .
```

Figure 2. Program segment to illustrate Symbol Table



ID	CLASS	TYPE	IN	OUT	OFFSET	NEXT
A	LOCAL	SCALAR	INT	—	∅	
B	LOCAL	SCALAR	CHAR	—	1	
X	LOCAL	FUNC.	CHAR	INT	2	
Z	LOCAL	SCALAR	INT	—	1	
A	GLOBAL	SCALAR	INT	—	∅	●
B	LOCAL	SCALAR	BOOL	—	2	●

IN = SCALAR TYPE, SET BASE TYPE, OR FUNCTION TYPE

OUT = SEQUENCE BASE TYPE OR FUNCTION OUTPUT TYPE

Figure 3. Symbol Table during compilation of figure 2

D. THE INTERPRETER

The Interpreter is a software simulator of the fetch-decode-execute cycle common in von Neumann computers. Programs are executed by fetching instructions from a code array and dispatching control to appropriate simulation routines.

Data is stored in the code array with the program. Scalar data is stored directly in an operand descriptor. A function is stored by keeping the code address of the body in the operand descriptor. A set or sequence is stored by placing a pointer to the structure in the operand descriptor. Figure 4 shows the data structures utilized.

Reference counts are maintained for sets and sequences. The storage manager uses these counts to decide whether a structure is reclaimable when released by the completion of an instruction cycle.

To maintain file bindings, the Interpreter creates a file control block (FCB) for the file and stores a pointer to the FCB in the operand descriptor of the bound sequence. File bindings are linked together so the Interpreter can find and close all files when the END operator is executed. The FCB structure contains a pointer back to the operand descriptor to enable the Interpreter to reference the KIND field of each binding during shutdown.

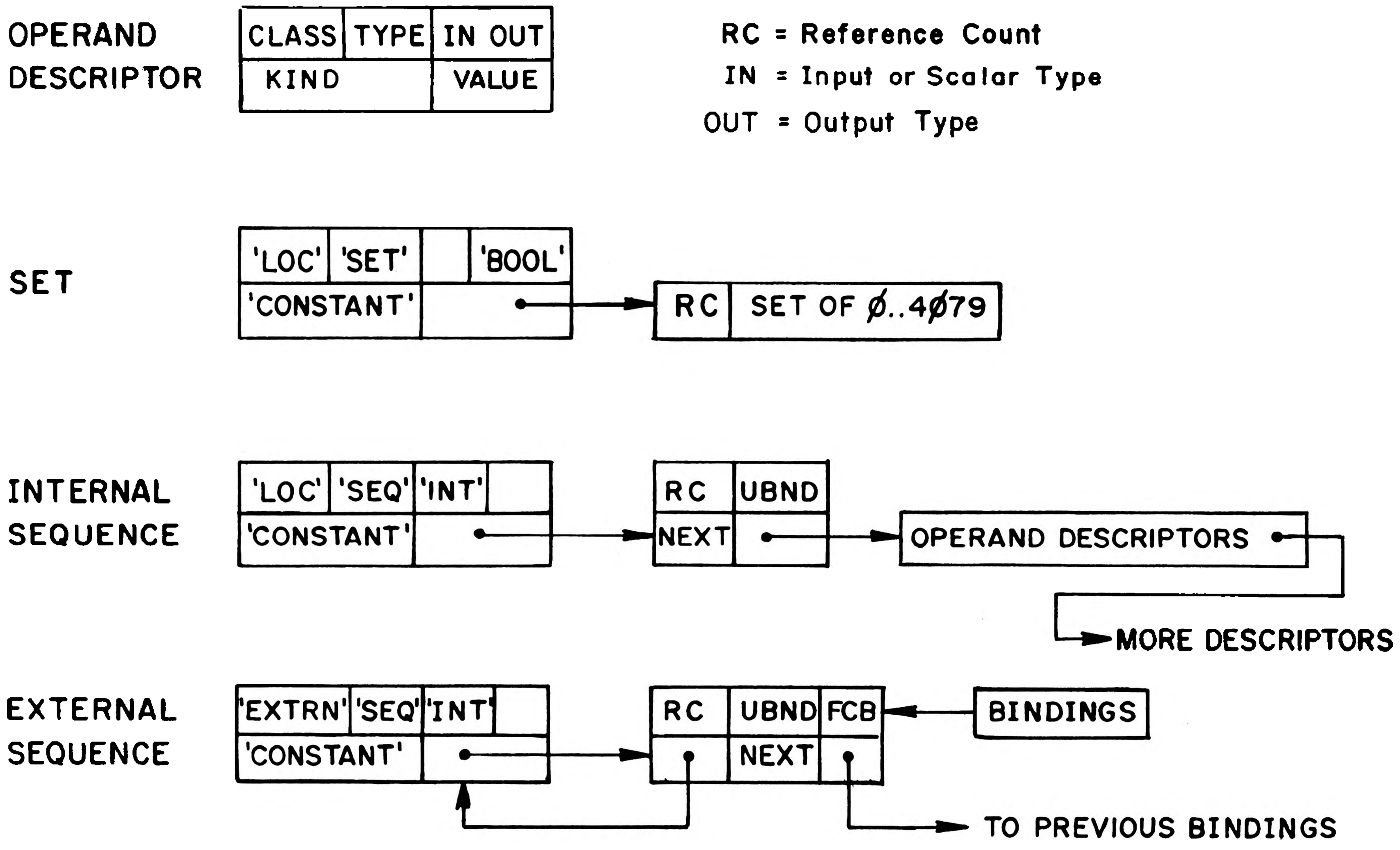


Figure 4. Interpreter data structures

Since the UCSD p-System restricts standard Pascal by disallowing file variables as fields in record structures, FCBs are stored by copying the contents of a Pascal file variable into a suitably-sized array in the FCB structure. When file access is required, the FCB is shifted back to a file variable, the access performed, and moved back.

IV. EXAMPLES

In this section examples of small programs written in Mini-Natural are presented.

Each figure is presented with two parts. Figure 'a' is the Mini-Natural source program. Figure 'b' is a dump of the file generated by the program when executed. The dump is produced by the SHOWSEQUENCE utility program.

A. POWER

Figure 5 shows a program designed to compute the value 6^5 by an algorithm attributable to Dijkstra [23]. To compute X^Y, X is repeatedly squared and Y halved while Y is even, then multiplied by X the remaining Y times. The reader can easily demonstrate that, for large Y, the algorithm requires fewer multiplications than multiplying X by itself Y times (4 as opposed to 8 for Y=8).

An interesting property about this program is that it virtually mirrors Dijkstra's solution in its syntax and semantic content. Indeed, Dijkstra's conceptions proved a fundamental influence in the design of Natural.

Note two properties regarding this program. First, due to the scope rules, the assignment to A in the program body has no effect on the value of A in the definition of P (the nonlocal reference is bound before this assignment). Second, program output is generated by

```
>>  Mini-Natural program to compute the value of 6 to the 5th power
>>   using Dijkstra's algorithm

Let
    a  be  int  <- 6;
    b  be  int;
    f  be  pwrseq (char) <- external "output.data";

    p  be  func (int -> int)  <-     >> output <- 6**input
            Let
                x be int <- a;
                y be int <- input
            Do [
                output <- 1;
                while (y > 0 -> [
                   while (2 div y -> [x <- x*x;  y <- y/2]);
                   output <- output * x ;
                   y <- y-1])
               ]

Do
   [
    a <- 10;
    b <-  5;
    f <- "6 to the " || text (b) || "th power = " || text (p(b))
   ]
End
```

Figure 5a. The POWER program

```
Sequence filename? OUTPUT.DATA
Format: I)nt C)har B)ool ? C

6 to the 5th power = 7776
```

Figure 5b. Output from POWER

the assignment to the sequence F, bound to an external file by the EXTERNAL expression in the declaration of F.

B. HAMMING

Figure 6 is a variation of an example which Dijkstra credits to R. W. Hamming [23]. The program is to produce the first N values of the increasing sequence defined by the axioms:

Axiom 1: The value 1 is in the sequence.

Axiom 2: If x is in the sequence, so are F(x) and G(x), where F and G are increasing functions, F(x) > x and G(x) > x.

Axiom 3: Only values accountable to applications of axioms 1 and 2 belong to the sequence.

The algorithm operates by extending the sequence (which starts with the value 1) with MIN(F(x),G(x)) and computing the next value for the function whose value was added to the sequence.

The availability of the sequence data type obviates the need for the programmer to explicitly keep track of the length of the sequence. Also, the sequence can grow to any length, without modifying the program to declare a size bound.

```
>> Generate in increasing order a sequence of numbers,
>>  given the following three axioms:
>>
>>  Axiom 1:  The value 1 is in the sequence.
>>  Axiom 2:  If x is in the sequence, so are f(x) and g(x).
>>  Axiom 3:  The sequence contains only values attributable to
>>            applications of axioms 1 and 2.

Let
   q be pwrseq (int) <- external "seqout.data";

   F  be func (int -> int) <- [output <- input * 2];
   G  be func (int -> int) <- [output <- input * 3];

   N  be int <- 20;
   fi be int <- 1;      fx be int <- F(fi);
   gi be int <- 1;      gx be int <- G(gi)

Do
   [
     q <- <.1.>;                    >> "1 is in the sequence"
     while (q`ubnd ^= N ->          >> upper bound of N numbers
        [
         if (fx >= gx -> q <- q || <.gx.>;
             gx >= fx -> q <- q || <.fx.>   );
         while (fx <= q(q`ubnd) -> [fi <- fi+1; fx <- F(q(fi))]);
         while (gx <= q(q`ubnd) -> [gi <- gi+1; gx <- G(q(gi))])
        ])
   ]
End
```

Figure 6a. The HAMMING program

```
Sequence filename? SEQOUT.DATA
Format: I)nt C)har B)ool ? I

             0     1     2     3     4     5     6     7     8     9
             _     _     _     _     _     _     _     _     _     _
   0 |             1     2     3     4     6     8     9    12    16
   1 |     18     24    27
```

Figure 6b. Output from HAMMING

V. SUGGESTIONS FOR FUTURE RESEARCH

The Mini-Natural implementation discussed in this paper is an experimental system, yet it demonstrates that an abstract language such as Natural can be feasibly implemented.

The inclusion of the sequence data type does make programming easier by removing from the programmer the responsibility for defining low-level primitives to manipulate list structures. The ability to assign various representations to structured data types permits a flexibility found in few compiled languages.

The language shows room for further experimentation in several areas. For instance:

1. The correct semantics of the special values UNDEF, POSINF, and NEGINF is currently unresolved. Should POSINF represent machine infinity or "true" infinity? Should expressions such as NEGINF+POSINF yield UNDEF or zero? The author believes that attempting to algebraically define operations upon the special values would introduce a myriad of confusing exceptional conditions, not consistent with the design goal of simplicity.

2. Whether the special constants complicate program debugging is unknown. Results of undefined or

erroneous operations can propagate through a Natural program, where other languages generate runtime errors. Experience with VAL should prove helpful in this regard [3,24].

3. The type of the EOLINE constant is often inconvenient. Choosing type char implies frequent use of the TEXT built-in function to concatenate the end-of-line marker to a sequence, however choosing type pwrseq(char) complicates checking a single input character against the marker. A possible improvement might be to have the compiler choose type char or pwrseq(char), depending on the context in which EOLINE is used.

The compiler and interpreter were coded using straightforward algorithms, choosing clarity over efficiency when tradeoffs were necessary. Suggestions for improvements include:

1. The algorithms for handling sets and sequences can be improved to reduce the number of copy operations performed. For instance, the statement F <- F || G might be performed by modifying F rather than modifying a copy of F and assigning the copy to F. Reference counts can be used, as in Schwartz' SETL implementation

[25], to determine whether a structure can be modified, rather than copied, without affecting the values of other variables which potentially address that structure.

2. Mapping sequences to external storage is very inefficient. Work must be done on the sequence handling routines if this method of file access is to be feasible.

3. The file handling routines must be extended to support input-output on devices such as printers, consoles, etc. Certain operations need to be defined; since sequences can be read or written, how should input from the printer be handled!

4. The runtime representation of data, especially sets and sequences, should be carefully reviewed. More efficient access methods, such as hashing, may be appropriate when processing sets and function reassignments.

5. Post-compilation optimization could be included to move some code (especially set or sequence creation code) outside of loops and replace references to these, as well as multiple creations of the same object, with a reference to a

temporary created by the compiler/interpreter system.

A different representation of the compiled quads might increase runtime efficiency. Rather than using intermediate code operators and operands, which must be painstakingly decoded during interpretation, the compiler might instead produce a sequence of operation codes driving a stack-based machine. The operation codes could be jump-table indexes, so dispatching would entail only indirect jumps to runtime system routines. This technique (or a variant) is often used to implement FORTH [26].

BIBLIOGRAPHY

1. OS PL/I Checkout and Optimizer Compilers: Language and Reference Manual. IBM GC33-0009-4, 5th Ed. (October 1976).
2. Jensen, K. and N. Wirth. Pascal User Guide and Report. 2nd Ed., New York: Springer-Verlag, 1975.
3. McGraw, J. R. "The VAL Language: Description and Analysis." ACM Transactions on Programming Languages and Systems, 4,1 (January 1982), pp 44-82.
4. Arvind, Gostelow, K. P. and W. Plouffe. "The (Preliminary) ID Report." Tech. Rep. TR 114a, Department of Information and Computer Science, University of California Irvine, Irvine, California.
5. McCarthy, J., et al. LISP 1.5 Programmers Manual. Cambridge: MIT Press, 1965.
6. Backus, J. "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." Comm. ACM, 21,8 (August 1978), pp 613-641.
7. Iverson, K. A Programming Language. New York: John Wiley and Sons, 1962.
8. VS FORTRAN Application Programming: Language Reference. IBM GC26-3986-3, 4th Ed. (March 1983).
9. Hoare, C. A. R. "Hints on Programming Language Design." In Tutorial: Programming Language Design, New York: IEEE, 1980, pp 43-51.

10. Myers, G. Software Reliability. New York: John Wiley and Sons, 1976.

11. Pratt, T. W. Programming Languages: Design and Implementation. Englewood Cliffs, NJ: Prentice-Hall, 1975.

12. Dijkstra, E. "GOTO Statement Considered Harmful." Comm. ACM, 11,3 (March 1968), pp 147-148.

13. Knuth, D. E. "Structured Programming with GOTO Statements." In Tutorial: Programming Language Design, New York: IEEE, 1980, pp 104-144.

14. Aho, A. and J. Ullman. Principles of Compiler Design. Reading, MA: Addison-Wesley, 1979.

15. Wulf, W., D. Russell, and A. Habermann. "BLISS: A Language for Systems Programming." Comm. ACM, 14,12 (December 1971), pp 780-790.

16. Wirth, N. Algorithms + Data Structures = Programs. Englewood Cliffs, NJ: Prentice-Hall, 1976.

17. Sager, T. "The Natural Language Report." To appear in 1984.

18. Sager, T. "Parallelism In the Language Natural." Rep. CSc-84-2, Department of Computer Science, University of Missouri-Rolla, Rolla, Missouri.

19. Schwartz, J. T. On Programming: An Interim Report on the SETL Project. I, New York: Courant Institute, 1976.

20. London, R., M. Shaw, and W. Wulf. "Abstraction and Verification In Alphard: Defining and Specifying Iteration and Generators." In _Tutorial: Programming Language Design_, New York: IEEE, 1980, pp 145-155.

21. Schwartz, J. T. "Automatic Data Structure Choice In A Language of Very High Level." _Comm. ACM_, 18,12 (December 1975), pp 722-728.

22. Calingaert, P. _Assemblers, Compilers, and Program Translation_. Potomac, MD: Computer Science Press, 1979.

23. Dijkstra, E. _A Discipline of Programming_. Englewood Cliffs, NJ: Prentice-Hall, 1976.

24. Wetherell, C. S. "Error Data Values in the Data-Flow Language VAL." _ACM Transactions on Programming Languages and Systems_, 4,2 (April 1982), pp. 226-238.

25. Schwartz, J. T. _On Programming: An Interim Report on the SETL Project_. II, New York: Courant Institute, 1973.

26. Loeliger, R. G. _Threaded Interpretive Languages_. Peterborough, NH: BYTE Publications, 1981.

VITA

Alan Lyle Sparks was born on June 26, 1960 in Fulton, Missouri. He received his primary and secondary education in Kingdom City, Missouri. He received a Bachelor of Arts degree in Computer Science from Westminster College, Fulton, in May 1982.

Alan has been enrolled in the Graduate School of the University of Missouri-Rolla, in Rolla, Missouri, since August 1982, and has held a graduate teaching assistantship for the full time of his enrollment.

APPENDIX A

SYNTAX SUMMARY OF MINI-NATURAL

```
program:
     letstmt end

letstmt:
     let declarations do simplestmt

simplestmt:
     ifstmt
     whilestmt
     assignstmt
     compoundstmt

compoundstmt:
     [ simplestmt { ; simplestmt } ]

declarations:
     declaration { ; declaration }

declaration:
     vardeclaration
     typedeclaration

vardeclaration:
     scalarvar be scalartype [ <- expr4 ]
     structuredvar be structuredtype [ <- expr4 ]
     structuredvar be structuredtype [ <- external expr4 ]
     structuredvar be structuredtype [ <- stmt ]
     variable { , variable } be type

typedeclaration:
     scalartypevar be type scalartype
     structuredtypevar be type structuredtype

scalartype:
     int
     bool
     char
     scalartypevar

structuredtype:
     pwrseq ( scalartype )
     pwrset ( scalartype )
     func ( scalartype -> scalartype )
     structuredtypevar
```

```
type:
     scalartype
     structuredtype

scalartypevar:
     variable

structuredtypevar:
     variable

ifstmt:
     if ( clause { ; clause } [ else simplestmt ] )

whilestmt:
     while ( clause { ; clause } )

clause:
     expr4 -> simplestmt

assignstmt:
     variable <- expr4
     variable <- stmt

stmt:
     letstmt
     ifstmt
     whilestmt
     compoundstmt

expr4:
     expr3 +  expr3
     expr3 -  expr3
     expr3 ⊤  expr3
     expr3 ⊤| expr3
     expr3 @  expr3
     expr3 ∑  expr3

expr3:
     expr2 *   expr2
     expr2 /   expr2
     expr2 mod expr2
     expr2 &   expr2

expr2:
     exprl inop  exprl
     exprl divop exprl
     exprl relop exprl

inop:
     in
     *in
```

```
divop:
     div
     ^div

relop:
      <     >     =     <>     <=     >=     <=>
     ^<    ^>    ^=    ^<>    ^<=    ^>=    ^<=>

exprl:
     variable
     constant
     ( expr4 )
     bifname ( expr4 )
     unaryop exprl
     { expr4 { , expr4 } }
     <. expr4 { , expr4 } .>
     textstring

variable:
     identifier
     identifier ( expr4 )
     identifier ^ attribute
     input
     output

constant:
     {}
     <..>
     eoline
     untypedconst
     intconst
     boolconst
     charconst

intconst:
      [ sign ] digit { digit }

sign:
     +
     -

boolconst:
     true
     false

charconst:
     'char

untypedconst:
     undef
     posinf
     neginf
```

```
attribute:
     ubnd
     next

bifname:
     int     char      bool       text      pred
     succ    choose    readint    readbool

unaryop:
     +
     -
     *
     _

textstring:
     " { char } "

letordig:
     letter
     digit

identifier:
     letter { letordig }
```

APPENDIX B

INTERMEDIATE CODE OPERATORS

NILARY OPERATORS:

```
[DATA]
     Function:    Mark data storage location.
     Mask:        Not used.

[END]
     Function:    Terminate execution, close files.
     Mask:        Not used.

[RETURN]
     Function:    Pop return address into PC.
     Mask:        Not used.
```

UNARY OPERATORS:

```
[ASSIGN]
     Function:    Copy operandl to destination.
     Mask:        Not used.
     Operandl:    Register, constant, or address.
     Destination: Register or address

[BIND]
     Function:    Bind external file to destination
                    sequence.
     Mask:        Not used.
     Operandl:    PWRSEQ(CHAR) containing filename.
     Destination: Address.

[COMPLEMENT]
     Function:    Complement of set operandl.
     Mask:        Not used.
     Operandl:    Address.
     Destination: Address.

[MINUS]
     Function:    Integer negation of operandl.
     Mask:        Not used.
     Operandl:    Register, constant, or address.
     Destination: Register or address.
```

[NOT]

Function:	Boolean NOT of operand1 and operand2.
Mask:	Not used.
Operand1:	Register, constant, or address.
Destination:	Register or address.

[STARTSEQ]

Function:	Create sequence containing value of operand1.
Mask:	Not used.
Operand1:	Register, constant, or address.
Destination:	Address.

[STARTSET]

Function:	Create set containing value of operand1.
Mask:	Not used.
Operand1:	Register, constant, or address.
Destination:	Address.

BINARY OPERATORS:

[ADD]

Function:	Integer addition of operand1 and operand2.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address
Destination:	Register or address.

[ADDTOSEQ]

Function:	Append value of operand2 to sequence operand1.
Mask:	Not used.
Operand1:	Address.
Operand2:	Register, constant, or address.
Destination:	Address.

[ADDTOSET]

Function:	Include value of operand2 in set operand1.
Mask:	Not used.
Operand1:	Address.
Operand2:	Register, constant, or address.
Destination:	Address.

[AND]

Function:	Boolean AND.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[ANDNOT]

Function:	Boolean ANDNOT (x AND (NOT y)) of operand1 and operand2.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[ASSONFALSE]

Function:	Copy operand1 to destination if value of operand2 is zero.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[ATTRIB]

Function:	Return attribute operand2 of sequence operand2.
Mask:	Not used.
Operand1:	Address.
Operand2:	UBND or NEXT.
Destination:	Register or address.

[CAT]

Function:	Concatenate sequences operand1 and operand2.
Mask:	Not used.
Operand1:	Address.
Operand2:	Address.
Destination:	Address.

[CMPFUNC]

Function:	Compare functions operand1 and operand2.	
Mask:	1 = not comparable	^<=>
	6 = not equal	<>
	8 = equal	=
	14 = comparable.	<=>
Operand1:	Address.	
Operand2:	Address.	
Destination:	Register or address.	

[CMPSCALAR]

Function: Compare scalars operand1 and operand2.

Mask:

1	= not comparable	^<=>
2	= greater-than	>
4	= less-than	<
6	= not equal	<>
8	= equal	=
10	= greater-than or equal	>=
12	= less-than or equal	<=
14	= comparable	<=>

Operand1: Register, constant, or address.
Operand2: Register, constant, or address.
Destination: Register or address.

[CMPSEQ]

Function: Compare sequences operand1 and operand2.
Mask: Same as CMPSCALAR.
Operand1: Address.
Operand2: Address.
Destination: Register or address.

[CMPSET]

Function: Compare sets operand1 and operand2.

Mask:

1	= not comparable	^<=>
2	= proper superset	>
4	= proper subset	<
6	= not equal	<>
8	= equal	=
10	= superset	>=
12	= subset	<=
14	= comparable	<=>

Operand1: Address.
Operand2: Address.
Destination: Register or address.

[DIFF]

Function: Difference of sets operand1 and operand2.
Mask: Not used.
Operand1: Address.
Operand2: Address.
Destination: Address.

[DIVIDE]

Function: Integer division of operand1 by operand2.
Mask: Not used.
Operand1: Register, constant, or address.
Operand2: Register, constant, or address.
Destination: Register or address.

[DIVIDES]

Function:	Test if operand1 evenly divides operand2.
Mask:	1 = x DIV y; 0 = x ^DIV y.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[EVAL]

Function:	Evaluate operand1 with argument operand2.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[IN]

Function:	Membership of operand1 in set operand2.
Mask:	1 = x IN y; 0 = x ^IN y.
Operand1:	Register, constant, or address.
Operand2:	Address.
Destination:	Register or address.

[INTSECT]

Function:	Intersection of sets operand1 and operand2.
Mask:	Not used.
Operand1:	Address.
Operand2:	Address.
Destination:	Address.

[MODULO]

Function:	Modulo division of operand1 by operand2.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[MULT]

Function:	Integer multiplication of operand1 and operand2.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[OR]

Function:	Boolean OR of operand1 and operand2.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[SUBT]

Function:	Integer subtraction of operand2 from operand1.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

[SYMDIFF]

Function:	Symmetric difference of sets operand1 and operand2.
Mask:	Not used.
Operand1:	Address.
Operand2:	Address.
Destination:	Address.

[UNION]

Function:	Union of sets operand1 and operand2.
Mask:	Not used.
Operand1:	Address.
Operand2:	Address.
Destination:	Address.

[XOR]

Function:	Boolean XOR of operand1 and operand2.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Register or address.

TRINARY OPERATORS:

[ASSATTRIB]

Function:	Assign value of operand2 to attribute operand1 of destination.
Mask:	Not used.
Operand1:	UBND or NEXT.
Operand2:	Register, constant, or address.
Destination:	Address.

[ASSCMPNT]

Function:	Assign value of operand2 to component operand1 of destination.
Mask:	Not used.
Operand1:	Register, constant, or address.
Operand2:	Register, constant, or address.
Destination:	Address.