REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-82-009 | 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) A Simple, Natural Notation for Applicative Languages	5. TYPE OF REPORT & PERIOD COVERED Techincal Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(*s*) Bruce J. MacLennan	8. CONTRACT OR GRANT NUMBER(*s*)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N: RR000-01--10 N0001482WR20043
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940	12. REPORT DATE September 1982
	13. NUMBER OF PAGES 16
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) Chief of Naval Research Arlington, Virginia 22217	15. SECURITY CLASS. (*of this report*) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

Notation, Applicative Languages, Functional Programming, Relational Programming, Logic Programming, PROLOG, Relational Databases, LISP.

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

Many non-specialists are intimidated by the mathematical appearance of most applicative, functional, and very-high-level languages. This report presents a simple notation that has an unintimidating, natural-language appearance and that can be adapted to a variety of languages. The paper demonstrates its use as an alternate syntax for LISP, PROLOG, Backus' FP, relational programming, and relational database retrievals. The grammar's eight productions can be handled by a simple recursive-descent parser.

A SIMPLE, NATURAL NOTATION FOR APPLICATIVE LANGUAGES*

B. J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

1. Introduction

Many non-specialists are intimidated by the mathematical appearance of most applicative and very-high-level languages. Mathematical notations have distinct manipulative advantages, some of which I have discussed in MacLennan (1979). Unfortunately the widespread use of advanced languages may be limited by their excessive use of mathematical notations. This paper presents a simple notation that has an unintimidating, natural-language appearance and that can be adapted to a variety of languages.

I must stress that I am not suggesting that this notation constitutes natural language programming. This notation is very far indeed from being even a subset of English, or any other natural language. However, the reader will see that with a proper choice of vocabulary the notation can be quite readable.

I must also stress that this notation is not in itself a programming language. It is more accurate to describe it as a

---

* Work described in this report was supported in part by the Office of Naval Research under contract number N00014-82-WR-20162.

syntactic framework that can be adapted to a number of specific contexts by a proper choice of vocabulary. The figures in this paper demonstrate its use as an alternate syntax for LISP, logic programming, functional programming, relational programming, and relational database operations.

2. Syntax

A natural, readable notation results from combining non-symbolic operator names with a right-associative infix syntax, and comma and colon rules that suppress many parentheses. Of course, some of the manipulative advantages of a mathematical notation are lost.

Briefly, the syntax is as follows: All identifiers are divided into three classes: niladic (x, y, z, in the following examples), monadic (f, g), and dyadic (p, q, r). Monadic applications, whether functions or predicates, are written "f x", "f g x", etc. These associate to the right, hence "f g x" means "f(g x)". Dyadic applications, whether functions or relations, are written with a right-associative, infix syntax. That is, "x p y q z" means "x p (y q z)". Monadic applications are more binding than dyadic applications; hence, "f x p g y" means "(f x) p (g y)". Operations that accept more than two operands are expressed by using a list building (or argument combining) operation. For example, if the operation "y with z" produces the pair (y,z), then the triadic operation p can be applied by "x p y with z".

Commas and colons can be used to eliminate many parentheses. A comma is equivalent to a right parenthesis. The corresponding left parenthesis is at the nearest preceding colon, or at the beginning of the expression, if there is no preceding colon. Hence, "x p y, q z" means "(x p y) q z" and "x p: y q z, r w" means "x p (y q z) r w", which by right-associativity means "x p ((y q z) r w)".

Since the parsing of expressions is determined by the classification of identifiers into niladic, monadic, and dyadic, it is not possible to directly use a monadic or dyadic identifier as the argument to another application. To do this it is necessary to convert the monadic or dyadic identifier into a niladic identifier by quoting it. For example, the inverse of the dyadic identifier plus must be written

```
inverse 'plus'
```

The formal grammar for this notation is in the appendix.

Figure 1 shows the natural notation adapted to LISP. The particular vocabulary choices shown are typical. The following two figures show a program in conventional LISP notation and in the natural notation. The remaining figures compare other mathematical and symbolic notations to the natural notation.

3. References

[1] MacLennan, B. J. Observations on the Differences Between

Formulas and Sentences and their Application to Programming Language Design, *SIGPLAN Notices* *14*, 7, (July 1979), pp. 51-61.

Appendix: Grammar for Natural Notation.

```
sentence          =    clause.
clause            =    term [predicate]
                  +    phrase, predicate
predicate         =    infix term [predicate]
                  +    infix: clause
phrase            =    simple-phrase
                  +    phrase, infix simple-phrase
simple-phrase     =    term [infix simple-phrase]
term              =    nilad
                  +    "(" clause ")"
                  +    prefix term
                  +    'monad'
                  +    'dyad'
                  +    constant
infix             =    dyad
                  +    "{" clause "}"
                  +    prefix infix
prefix            =    monad
                  +    "[" clause "]"
```

Natural Notation	LISP
"X F Y with Z" means B	(defun F (X Y Z) B)
"X F Y" means B	(defun F (X Y) B)
"F X" means B	(defun F (X) B)
C if B, else D	(cond (B C) (T D))
"X" means Y, below B	(let ((X Y)) B)
first X	(car X)
rest X	(cdr X)
second X	(cadr X)
third X	(caddr X)
X with Y	(cons X Y)
X is Y	(eq X Y)
atom X	(atom X)
null X	(null X)
number X	(numberp X)
X append Y	(append X Y)
X search Y	(assoc X Y)

Figure 1. Comparison of Natural Notation and LISP

```
(defun eql (x y)
  (or (and (atom x) (atom y) (eq x y))
      (and (not (atom x)) (not (atom y))
           (eql (car x) (car y))
           (eql (cdr x) (cdr y)) )) )
```

Figure 2. Equal Function in LISP

```
"X equals Y" means:
  atom X and atom Y and X is Y, or
  not atom X and not atom Y and:
  first X equals first Y, and
  rest X equals rest Y.
```

Figure 3. Equal Function in Natural Notation

```
Isa (John, human).
Gives (John, book, Mary).
Gives (John, book, x)  ←  Likes (John, x).
Likes (w,x)  ←  Gives(w,y,x), Likes(w,y).
```

Figure 4. Logic Program in Usual Notation

```
John isa human.
John gives book to Mary.
John gives book to one., if John likes one.
One likes another, if:
  one gives gift to another, and one likes gift.
```

Figure 5. Logic Program in Natural Notation

```
Def IP = (/+)·(∝ X )·trans.

Def MM = (∝ ∝ IP)·(∝ distl)·[1, trans·2]
```

Figure 6. Functional Program in Backus Notation

```
Inner-product means
  transpose then repeat times then reduce-by plus.

Matrix-multiply means:
  first combine second then transpose,
  then repeat distribute-left
  then repeat repeat inner-product.
```

Figure 7. Functional Program in Natural Notation

f\$R = f^{-1}.R.f

rightsib = T^{-1}\$(Id¦¦(+1))

next = move.total [while(non.dom rightsib, parent); rightsib]

prev = move.total
    [while(non.dom rightsib^{-1}, parent); rightsib^{-1}]

remove(L) = L := subtree N; excise
subtree(n) = (m ¦ m X ints) → T
    where m = subnodes n
reach = (img T).(X ints)

excise = T := T <> non.subnodes N ¦ (T^{-1}N, N, NT N)

replace(L) = T := (T^{-1}N : first L ¦ L) / T

Figure 8. Part of Syntax Directed Editor in Relational Notation

```
"Function map structure" means
  function then structure then inverse function.
"Right-sibling" means
  inverse tree map identity parallel something plus 1.
"Move-next" means parent do-while non domain right-sibling,
  then right-sibling, apply total then move.
"Move-previous" means
  parent do-while non domain inverse right-sibling,
  then inverse right-sibling, apply total then move.

"Remove-from buffer" means:
  buffer becomes subtree of current-node, then excise.
"Subtree a-node" means:
  tree if-in the-subnodes combine the-subnodes cross integers,
    where the-subnodes means subnodes of a-node.
"Reach" means:  something cross integers, then image tree.

"Excise" means tree becomes
  tree restrict non subnodes of current-node
  combine: current-node apply inverse tree,
  connect current-node connect non-term of current-node.

"Replace-from buffer" means tree becomes:
  current-node apply inverse tree, maps-to first buffer,
  combine buffer, extend tree.
```

Figure 9. Part of Syntax Directed Editor in Natural Notation

$\{(F.COMPANY): F \in FORESTS \wedge F.SIZE>1000\}$

$\{(F.COMPANY,F.FOREST): F \in FORESTS \wedge F.LOC='CALIFORNIA'\}$

$\{(F.SIZE,F.LOC): F \in FORESTS \wedge$
  $\exists\ T \in TREE\ (T.SPECIES='CEDAR' \wedge T.FOREST = F.FOREST)\}$

$\{(F.SIZE,T.TREENUM): F \in FORESTS \wedge T \in TREE \wedge$
  $T.FOREST = F.FOREST \wedge T.SPECIES = 'CEDAR'\}$

Figure 10. Relational Database Retrievals in Conventional Notation

Company F whenever: F in forests, and size F > 1000.

Company F with forest F, whenever:
  F in forests, and location F is "California".

Size F with location F, whenever: F in forests,
  and: T in trees, exists:
    species T is "cedar", and forest T is forest F.

Size F with tree-number F, whenever:
  F in forests, and T in trees, and
  forest T is forest F, and species T is "cedar".

Figure 11. Relational Database Retrievals in Natural Notation

```
(defun eval (e a)
  (cond
   ((and (atom e) (numberp e))  e)
   ((atom e)  (assoc e a))
   ((eq (car e) 'quote)  (cadr e))
   ((eq (car e) 'cond)  (evcon (cdr e) a))
   (T  (apply (car e) (evargs (cdr e) a) a) )) )

(defun evcon (L a)
  (cond
   ((eval (caar L) a)  (eval (cadar L) a))
   (T  (evcon (cdr L) a)) ))

(defun evargs (x a)  (mapcar (bu (rev 'eval) a) x))

(defun apply (f x a)
  (cond
   ((eq f 'car)  (car (car x)) )
   ((eq f 'cdr)  (cdr (car x)) )
   ((eq f 'atom) (atom (car x)) )
   ((eq f 'null) (null (car x)) )
   ((eq f 'cons) (cons (car x) (cadr x)) )
   ((eq f 'eq)   (eq (car x) (cadr x)) )
   (T (let ((L (eval f a) ))
           (let ((LE (mapcar 'list (cadr L) x) ))
                (eval (caddr L) (append LE a)) )) )) )
```

Figure 12. LISP Universal Function in LISP

```
"Names evaluate form" means:
  form if (atom form and number form), else:
  names search form if atom form, else:
  second form if first form is "quote", else:
  names do-conditional rest form, if first form is "cond", else
  names apply first form with names evaluate-list rest form.

"Names do-conditional pairs" means:
  names evaluate second first pairs,
    if names evaluate first first pairs,
  else names do-conditional rest pairs.

"Names evaluate-list forms" means:
  nil if null forms, else:
  names evaluate first forms,
  with names evaluate-list rest forms.
```

Figure 13. LISP Universal Function in Natural Notation (Part 1)

```
"Names apply function with actuals" means:
  first first actuals if function is "car", else:
  rest first actuals if function is "cdr", else:
  atom first actuals if function is "atom", else:
  null first actuals if function is "null", else:
  first actuals with second actuals, if function is "cons", else:
  first actuals is second actuals, if function is "eq", else:
  names apply-user function with actuals.

"Names apply-user function with actuals" means:
  lambda-expression means names evaluate function, below:
  bound-variables means second lambda-expression, below:
  bound-variables pair-with actuals, append names,
  evaluate third lambda-expression.

"Names pair-with values" means:
  nil if null names, else:
  first names with first values,
  with rest names pair-with rest values.
```

Figure 14. LISP Universal Function in Natural Notation (Part 2)

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22314

Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93940

Office of Research Administration 1
Code 012A
Naval Postgraduate School
Monterey, CA 93940

Chairman, Code 52Hq 40
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940

Professor Bruce J. MacLennan, Code 52Ml 12
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940

Jim Bowery 1
Viewdata Corp. of America, Inc.
Suite 305
1444 Biscayne Blvd
Miami, FL 33132

Dr. R. B. Grafton 1
Code 433
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217