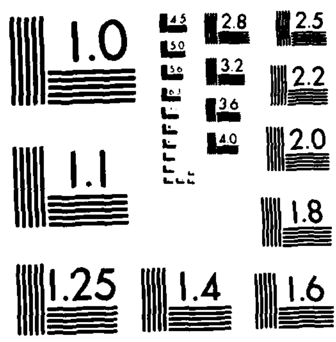
AD-A171 664 A COLLECTION OF ALFA EXAMPLES(U) CARNEGIE-MELLON UNIV PITTSBURGH PA A N HABERMANN 05 NOV 81 DAAB07-82-C-J173 1/1

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART

AD-A171 664

A Collection of Alfa Examples

A. N. Habermann
Carnegie-Mellon University
Pittsburgh, Pa 15213

Contract DAAB-07-82-C-J173

DTIC
ELECTE
SEP 5 1986
S D
B

Abstract

Alfa is a functional programming language used in the context of Programmatics, a programming environment in which one can apply rewriting rules and in which one can express equivalence of functional expressions. Alfa provides strong typing, information hiding, suitable infix notations, overloading and modularization. The examples are presented without an explanation of the Alfa language. Detailed, though partly obsolete, information is found in the documents "Notes on Programmatics" Part I and Part II.

This work is supported by the Research Division of Coradcom, FtMonmouth, N.J.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

86 7 7 186

Table of Contents

1 Calendar. 1
2 Vectors. 2
3 Local Names. 3
I. Alfa Grammar. 7
II. Standard Functions. 8

PER LETTER
A-1

1 Calendar.

The goal of the example is a program that finds the weekday of a given date within the period 1900 through 2099.

```
module CALENDAR :=

      weekday    := (SUN. MON, TUE. WED. THU. FRI, SAT)
      month      := (JAN, FEB. MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)

      day        := nat                for  id ∈ (1 .. 31)
      year       := nat                for  id ∈ (1900 .. 2099)
      date       := (year, month, day)

      Weekdayof : date → weekday      :=
                 weekday # (((id + id%4) ∘ (first — Base ∘ ftail) + Map ∘ ftail + last) % 7)

where

      Map        : month → int        :=     (1 4 3 6 1 4 6 2 5 0 3 5) # (month ? id)

      Base       : month → year       :=     id = JAN or id = FEB ⇒ 1901 ; 1900

end CALENDAR
```

seq # n — returns the nth element of "seq". This function corresponds to array indexing: "seq[n]".

seq ? elem — returns the index of the leftmost occurrence of "elem" in "seq". (For new domains. it corresponds to the functions "POS" in Ada and "ord" in Pascal.)

n % p — is the remainder of the integer division of integer n by natural p.

2 Vectors.

An example of a generic module.

module VECS(obj) :=

```
    vec        := (obj . . )
    Binop      := (obj, obj) → obj

    Vecop ! Binop :  (vec, vec) → vec   :=        eql ∘ αlen ⇒ αID ∘ // ; Error
```

end VECS

An example of the use of generic module "VECS".

module REALVECS :=

```
    use  REALS, VECS(real)

    " + "     : (vec, vec) → vec       :=         Vecop ! REALS." + "
    "•"       : (vec, vec) → vec       :=         Vecop ! REALS."•"

              — — — — —
```

end REALVECS

seq // seq represents the function "transpose".

3 Local Names.

An example of the problems that may arise from the use of local names.

Problem Statement.

Determine the multiplicity of the (value of the) first element of a non-empty sequence of integer numbers.

First Solution.

Compare the first element with each element of the input sequence and count the number of times you find equality.

Program 1.

```
Firstmultcount : (int . . ) → nat  :=   + ∘ α(first = id ⇒ 1 ; 0)
```

The functional "α" is what is called MAPCAR in Lisp. Thus, the equality comparison with the first element is applied to all elements of the input sequence and each element is mapped *into 1 or 0* depending on the outcome TRUE or FALSE. The final result is obtained by adding all elements of the sequence of zeros and ones. Addition is a dyadic function of a special class of functions whose output type is the same as that of the first element of the input sequence. These functions are extensible to non-empty sequences of arbitrary length. I envisage that the extension of such a dyadic function is automatically derived from the original dyadic definition by the language system so that there is no need for an explicit functional "REDUCE" as represented in APL by the symbol "/". The extended function is left associative so that the extension of dyadic functions such as subtraction, division or remainder have the desired result for input sequences of more than two elements.

Now we introduce a local name for the first element and rewrite the program.

Program 2.

```
Firstmultcount : (int . . ) → nat  :=
  let u := first in  + ∘ α(u = id ⇒ 1 ; 0)
```

This program works the same way as the first program with the slight difference that the first element is remembered in the local object named "u".

Second Solution.

Distribute the first element over all elements of the input sequence, forming a sequence of pairs (first, y) where y is one of the elements of the input sequence. Count the number of equal pairs in this sequence of pairs.

Program 3.

Firstmultcount : (int . .) → nat := + ∘ α(eql ⇒ 1 ; 0) ∘ (first *> id)

The symbol "*>" represents distribution from left to right. The program distributes the first element, applies the equality test to every pair and adds the number of resulting ones and zeros.

Program 4.

Firstmultcount : (int . .) → nat :=
let u := first **in** + ∘ α(eql ⇒ 1 ; 0) ∘ (u *> id)

This program works basically the same as the third program.

The question is whether or not *there is much difference* between the first two programs and the last two programs. Before turning to the discussion on the next page, you should form your opinion about the four programs on this page.

Discussion.

Do you believe that all four programs are correct?

If you do, you must have another way of attaching meaning to functional programs than I do. In my opinion, Program 3 is correct, but Program 1 is incorrect. The problem with Program 1 is the application of the function

(first = id ⇒ 1 ; 0)

to each individual element of the input sequence. It is alright to apply function "id" to an integer number (that is what the elements of the input sequence are), but it is not alright to apply function "first" to an integer number. This function is defined for non-empty sequences, but not for atomic objects. Of course, we meant to apply function "first" only once to the input sequence and then use the result for comparison with the elements of the input sequence. But that is not what we get. With the implementation of Program 1, function "first" is indeed applied to every individual element of the input sequence which causes the erroneous application of function "first" to an atomic object. It is clear that a similar error does not occur in Program 3, because there function "first" is applied exactly once to the entire input sequence.

There is no problem in interpreting Program 4, because the local object "u" is used only once. However, what about Program 2? I can see two possible interpretations: one that makes the program correct and another that makes the program incorrect. The first interpretation considers "u" to be a local constant that is computed only once (if needed) per execution of function "Firstmultcount". If you accept this interpretation, Program 2 is correct, because each element is now compared to a constant.

The second interpretation considers "u" to represent the expression on the righthand side of the definition symbol ":=". If you accept that interpretation, you agree that the program is incorrect, because it now has the same flaw as Program 1.

Note that the problem is not a matter of lazy evaluation, but one of interpreting the definition of the local object. The two different interpretations correspond in fact to the Algol concepts of "call-by-value" and "call-by-name".

I am not suggesting that one interpretation is better than the other. I also don't care much which interpretation is more commonly used than the other. The point I am trying to make is that the problem of being forced to choose and explain the semantics arises because of the introduction of local names. No explanation would be necessary if the concept of local names is not introduced at all. That, in my opnion, reduces the complexity of a language and avoids the likely occurrence of misunderstanding among programmers. You may have a trivial solution to the problem, or you may want to convince me that the added complexity is a price well worth to pay. I am listening.

I. Alfa Grammar.

```
Program          →   Modulesequence

Module           →   module Ident {(Generictypelist)} :=
                      {use Modnamelist}
                      Declsequence
                      {where Auxdeclsequence}
                     end Modname

Generictype      →   Ident

Decl, Auxdecl    →   Domaindecl | Classdecl | Fundecl | Formdecl

Domaindecl       →   Ident := NewDomaindef | Ident := Domaindef {for Predicate}
NewDomaindef     →   (Identlist)
Domaindef        →   {atom} Domain
Domain           →   Generictypename | Domainname | (Domainlist { . { . }} )

Classdecl        →   Ident :=  Classdef {for Predicate}
Classdef         →   Domain → Domain
Class            →   Classname | Classdef

Fundecl          →   Ident : Class := Expr
Formdecl         →   Ident ! Param : Class := Expr
Param            →   Domain | Class | (Paramlist { . { . }} )

Expr             →   Dyad { ⇒ Expr ; Expr}
Dyad             →   Term {Funsym Term}
Term             →   Factor { ∘ Factor}*
Factor           →   {Funal}* Primary { ! Primary}* | Formname ! Factor

Primary          →   Funname | Funsym | Const | (Exprlist)
Funal            →   α | β | γ
Funsym           →   + − * / % @ # $ ↑ & ~ | ? < ≤ = ≠ ≥ >
                     ≪ ≫ ∈ // <* *> ⊂ ⊃ ∩ ∪ ||
```

Shorthands are used for strings and ranges. E.g., "functional", (2 . . 17)

The functionals α, β and γ stand for "all", "filter" and "combine" resp.

II. Standard Functions.

Monadic Functions

Name	Specification	Operation
Identity	id : x → x	maps object into itself
Nil test	null : x → bool	TRUE iff input is the empty seq
Is atom	atom : x → bool	TRUE iff input is an atom
Is set	$: z → bool	TRUE iff multiplicity of every elem of input seq is one
Const function	*const* : x → *const*	returns that const on the domain of x
First element	fst : s → x	returns the first element of non-empty seq s
Second element	snd : (x y . .) → y	input must have at least two elems: returns second elem
Tail end	tl : s → z	all but the first element of non-empty seq s
Head part	hd : s → z	all but the last element of non-empty seq s
Last element	lst : s → x	returns the last element of non-empty seq s
Length	len : z → znat	returns the length of seq z
Make seq	seq : x → s	creates sequence with input as single element
Make range	run : nat → s	generate range 1 . . input
Make set	set : z → z	reduce multiplicity of elems of input seq to one
Reverse	rev : z → z	reverse the elements of input seq z
Shift left	left : z → z	move first elem (if any) to rear of input seq z
Shift right	right : z → z	move last elem (if any) to front of input seq z
Extremities	edge : z → z	form pair of first and last elem if input seq non-empty
Fold seq	fold : z → z	seq formed by repeatedly pealing off edges until empty
Unfold seq	unfold : z → z	remove all inner parentheses if input seq non-empty

Dyadic Functions

Name	Representation	Operation
Push elem	x ≫ z → s	add x as first elem to seq z
Append elem	z ≪ x → s	add x as last elem to seq z
Join seqs	z & z → z	concat seqs, preserving the order of the elems
Distr. left	(y .) <* x → ((y, x) .)	distribute x over all elems (if any) of seq z
Distr. right	x *> (y .) → ((x, y) .)	distribute x over all elems (if any) of seq z
Select	s # nat → x	select elem s[nat] from non-empty seq s
Collection	s # (nat .) → z	form seq of selected elems s[nat]
Replicate	nat # x → (x . .)	form seq by replicating x nat times
Rotate	z @ int → z	rotate input over int elem positions
Index	s ? x → znat	0 if x is not in seq s, else index of leftmost x in seq s
Equality	x = y → bool	TRUE iff equal atoms or both nil or corresp elems equal
Transpose	(x .) // (y .) → ((x, y) .)	form seq of pairs of corresponding elems
Remainder	z % (x . .) → z	remove all occurrences of elems x (if any) from input seq z
Split seq	z / (x . .) → z	split input seq z *every time you find one of the elems x*
Partition	z ‖ nat → z	partition input seq into slices of length nat
Union	z ∪ z → z	form the union of the input sets
Intersection	z ∩ z → z	form the intersection of the input sets
Remove elem	z — x → z	if x is in set z, remove it from set z
Insert elem	x +> z → s	if x is not in set z, add x as first elem of set z
Insert elem	z <+ x → s	if x is not in set z, add x as last elem of set z
Member	x ∈ s → bool	TRUE iff x is an elem of set s
Subset	z ⊂ z → bool	TRUE iff first set is a subset of second set
Include	z ⊃ z → bool	TRUE iff first set includes second set

END

10-86

DTIC