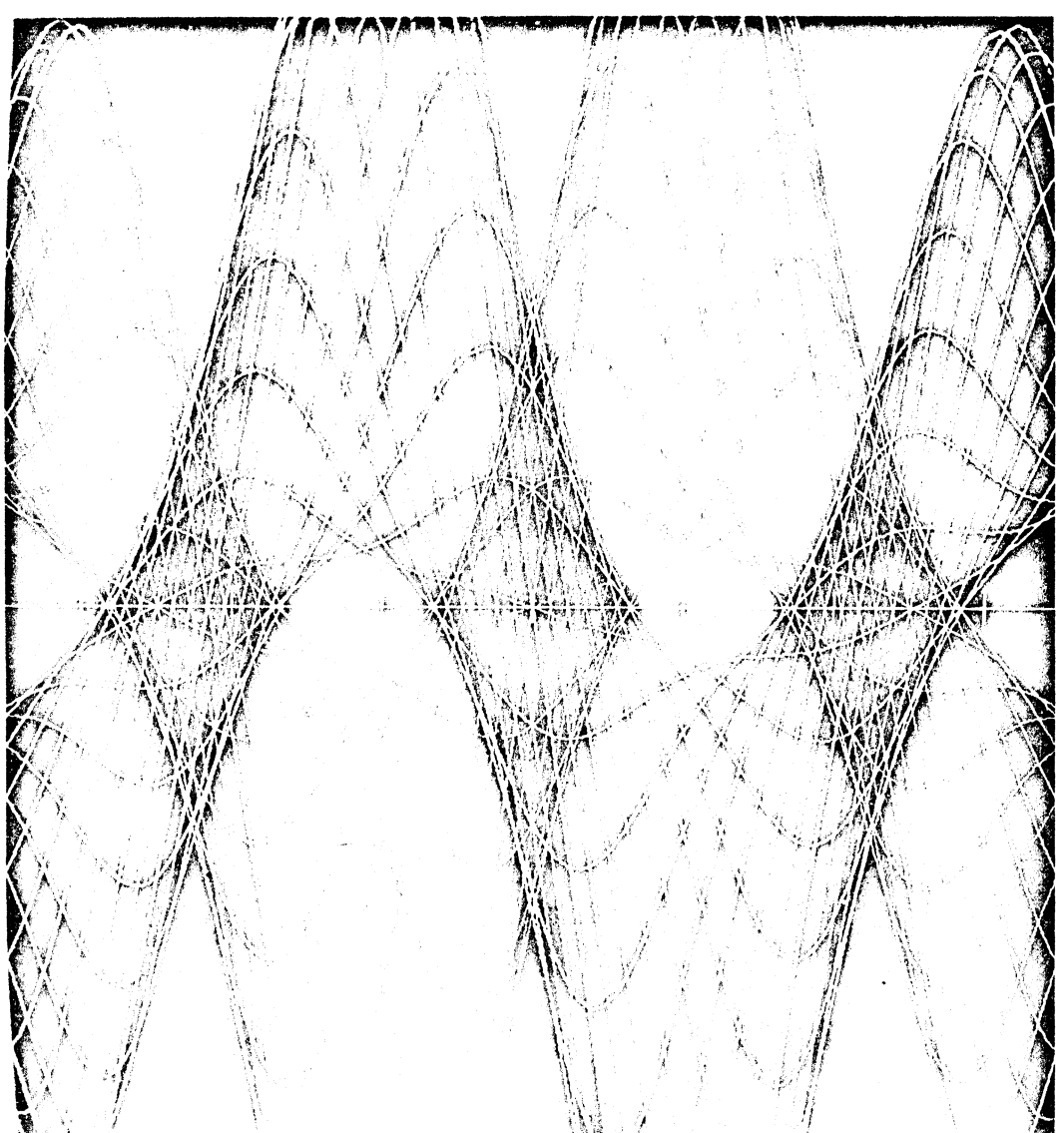

RC 2189

MCG - A FUNCTIONAL PROGRAMMING SYSTEM

August 29, 1968

IBM RESEARCH

MCG - A FUNCTIONAL PROGRAMMING SYSTEM

W. H. Burge
Research Division
Yorktown Heights, New York

ABSTRACT: The Mcg conversational programming system is described informally by listing a series of messages given to it together with its responses. The essential details of the implementation are then described using its own language.

RC 2189 (# 10969)
August 29, 1968
Programming and programming languages

INTRODUCTION

Mcg is a conversational programming system which was implemented as part of a project to investigate the implementation and use of programming languages which rely on the use of expressions rather than statements as their main linquistic unit.

The system is first described informally by listing a series of messages given to the machine, followed by its responses. Its implementation is then described more formally.

Two types of message may be given to the system:

(1) An expression such as 'a+b-c'.

(2) A definition such as 'def x=3'.

It responds to an expression by printing its value and stores a definition for future use. The expressions are construed as descriptions of objects of the following types:

(1) Integer

(2) Character

(3) List of Objects

(4) Basic Function

(5) Defined Function

Certain basic functions for arithmetic, for list construction and decomposition, and predicates for these five types are built in. The lists may be lists of any type of object and functions may be defined to operate on any type of object and produce any type of object.

2. LITERALS AND DEFINITIONS

```
2;
2

2+2;
4

[abcd];
ABCD

[abcd][abcd];
ABCDABCD
```

Mcg has the infixed operators +, -, /, *, >, <, = built into it and applicable to integers. Square brackets are used to quote a string of characters and the value of an expression composed by the juxtaposition of two character string expressions is the result of concatenating the two strings.

```
def x=2;
def s=[abcd];
```

These definition messages give rise to no machine response. The identifiers are paired with the value of the definiens expression for future use. e.g.

```
x;
2
```

```
s;
ABCD

x+x;
4

s s;
ABCDABCD
```

It is also possible to define functions for future use as follows:

```
def f(y)=[y+y];

f x;
4

def g(y)=[y y];

g s;
ABCDABCD
```

The square brackets have a second use in bracketing the body of the definiens of a function.

```
f(f x);
8

g(g s);
ABCDABCDABCDABCD
```

The operand position of an expression may be a compound expression. The juxtaposition of two expressions means functional application if the value of the operator expression is a function, if the value is a string it means concatenation.

```
let x=3 f x;
6

let s=[efgh]g s:
EFGHEFGH

let f x=[x+x] f 3:
6

let g y=[y y] g[efgh];
EFGHEFGH
```

An expression may be formed with an auxiliary definition introduced by 'let'. The definition is temporary and only holds during the evaluation of the expression which it qualifies.

```
'p;
p

'[;

';
'
```

A single character may be quoted by preceding it with a quote mark(').

```
2,2;
(2,2)

x,x;
(2,2)

s,s;
(ABCD,ABCD)
```

An expression which denotes a list may be written by separating the expressions by commas. The value of a list of expressions is a list formed from the values of the expressions.

```
def plus(x,y)=[x+y];

def ccat(x,y)=[x y];
```

Functions expecting a list of arguments may be de-

fined as above and used as follows.

```
plus(2,3);
5

ccat([abcd],[efgh]);
ABCDEFGH

def sq x=[x*x];

def v(x,y)=[sq x+sq y,sq x-sq y];

v(v(3,2));
194,144)
```

Functions like v which produce a list of results may be defined.

```
def max(x,y)=[y>x?y!x!];

max(2,6);
6
```

Conditional expressions have the format 'p?a!b!'. Its value is the value of a if the value of p is true, otherwise it is the value of b. The value of p is found before either a or b is evaluated.

```
def f=Δx[x+x];

f 2;
4
```

This is another way to define f and the message is equivalent to 'def f x=[x+x]'. The expression Δx[x+x] is an expression which denotes the function for doubling a number.

```
def pl x=[Δv[x+v]];

def eq x=[Δy[x=y]];

def gr x=[Δy[y>x]];

def ls x=[Δy[y<x]];

def q r=[Δ(x,y)[r x y?y!x!]];

def mx=(q gr);

def mn=(q ls);

mx(2,6);
6

mn(2,6);
2
```

Functions such as gr and ls above may appear as arguments of other functions and functions like mx and mn may be obtained by the application of a function to its arguments.

```
def i x=[x];

pl 2 2;
4

(pl 2)2;
4

(p(2))(2);
4
```

The function pl operates on an integer to produce a function for adding the integer to another. It is possible to write compound expressions in operator position of an expression. The bracketing associates to the left and brackets are only used for purposes of grouping.

```
def w f=[Δx[f x x]];

def b f=[Δg[Δx[f(g x)]]];

def s f=[Δg[Δx[f x(g x)]]];

s pl sq 3;
12

def phi f=[Δa[Δb[Δx[f(a x)(b x)]]]];
```

Here are some other functions which operate on, and produce, functions.

i is the identity function,

k produces constant functions,

w applies its argument f to its second argument duplicated.

```
i 3;
3

k 3 2;
3

w mt 2;
4

def sq=(w mt);
```

b is functional composition

```
3*2+4;
10

b(pl 4)(mt 2)3;
10
```

The function c reverses the two arguments of a function f. The function phi pl is the sum of two unary functions, and phi(phi pl) is the sum of two binary functions.

```
pl 2 3;
5

c i 3(pl 2);
5

phi pl sq (mt 3) 5;
40

phi(phi pl)pl mt 5 6;
41

def zero f=[Δx[x]];

def once f=[Δx[f x]];

def twice f=[Δx[f(f x)]];

def thrice f=[Δx[f(f(f x))]];
```

The functions apply their functional argument (f) zero, once, twice, or three times to their second argument(x).

```
twice sq 2;
16

thrice sq 2;
256

def rec z n=[Δf[Δx[n=0?x!z(n-1)f(f x)!]]];
```

The function z operates on an integer n to produce a function which applies its functional argument f, n times to its other argument (x). Thus we can write an expression which is sometimes written $f^n x$ (e.g. in $\sin^2+\cos^2 x$) as "z n f x'. The piece of syntax 'rec' is introduced in order to indicate that the occurrence of z in the definiens refers to the #

being defined and not a z which might have been previously defined.

```
z 2 sq 2;
16

z 3 sq 2;
256

def suc g=[Δf[Δx[f(g f x)]]];

suc(z 2)sq 2;
256

def d2 x=[Δy[Δz[z(k y)x]]];

d2 1 2(z 0);
1

d2 1 2(z 1);
2

def rec r a=[Δg[Δn[n=0?a!g n(r a g(n-1))!]]];
```

When r is applied to an initial value a and a function from two integers to an integer (g) it produces a function from integers to integers.

```
r 0 pl 4;
10

r 1 mt 5;
120
```

The function r is called the 'operator of primitive recursion' and a large number of functions can be cast into this form.

```
def second(x,y)=[y];

second(4,8);
8

def r1 a=[Δg[Δn[second(z n Δ(x,v)[x+1,g x y](1,a)]]];

r1 0 pl 4;
10

r1 1 mt 5;
120
```

Here is another definition of r defined in terms of the successor function rather than the predecessor.

```
def rec mu p=[Δn[p n?n!mu p(n+1)!]];

def insqrt n=[mu(Δm[sq(m+1)>n])0];

insqrt 67;
8
```

The value of mu p n is the first integer after n having the property p. If there is no such integer the corresponding program runs forever.

```
def y f=[let g h=[f(Δn[h h n])]g g];

def r2=(y Δs[Δa[Δg[Δn[n=0?a!g n(s a g(n-1))!]]]]);

r2 0 pl 4;
10

r2 1 mt 5;
120
```

The syntactic device 'rec' used for self referential functions can be eliminated by using the function y. This is shown above in which r2 is another definition of r.

3. LISTS

```
h(1,2,3,4);
```

```
1

t(1,2,3,4);
(2,3,4)

pfx 0(1,2,3,4);
(0,1,2,3,4)
```

The functions h, t and pfx are built in, h and t are applicable to lists,

h selects the first item of a list,

t selects all but the first item and pfx prefixes a new item to the front of a list. A null list is written '[]' and gives no printed response.

```
[];

def th n=[b h(z n t)];

th 3(0,1,2,3,4,5,6);
3
```

A predicate called 'null' tests whether a list is empty or not.

```
pfx 4[];
(4)

pfx 3(pfx 4[]);
(3,4)

3,4;
(3,4)
```

Strings of characters are represented as lists of characters and so these functions may also be applied to strings of characters. The printing conventions for lists are:- if the list is a list of characters the string is printed, if not then the list items are surrounded by parentheses and separated by commas.

```
'a,'b,'c,'d;
ABCD

pfx 'a[bcd];
ABCD

h[abcd];
A

t[abcd];
BCD

def rec length x=[null x?0!1+length(t x)!];

length[abcd];
4

length[a];
1

length[];
0

'a;
A

[a];
A

def u x=[pfx x[]];

3;
3

u 3;
(3)
```

A one list containing a character is not distinguished on printing from a character, but a one-list containing a non character is printed surrounded by parentheses.

```
def rec map f=[λl[null l?[]!pfx(f(h l))(map f(t l))!]].
```

```
This function applies its functional argument to all

the items of a list and produces a list of the results.

map sq(1,2,3,4,5);
(1,4,9,16,25)

map length([abc],[gh],[h],[l]);
(3,2,1,0)

def rec list1 a=[Δg[Δf[Δl[null l?a!g(f(h l))

                                (list1 a g f(t l))!]]]];

This function embodies a particular way of scanning

a list and it is possible to define functions of the

form:  take the head (h) of the list, apply f to it,

and apply g to the result and the result of applying

the same function to the tail (t) of the list e.g.

def length=(list1 0 p1(k l));

length[asdzfxgx];
8

def map f=[list1 [] pfx f];


map(p1 2)(1,2,3,4,5,6);
(3,4,5,6,7,8)

def app x=[Δy[list1 y pfx i x]];

This function concatenates two lists

app(1,2,3)(4,5,6);
(1,2,3,4,5,6)

def ccat=(list1 [] app i);

ccat([abc],[def],[ghi],[j]);
ABCDEFGHIJ

def sum=(list1 0 p1 i);

sum(12,23,34);
69

def prod=(list1 1 mt i);


prod(1,2,3,4,5);
120

def pofx x=[Δy[app y(u x)]];

pofx 1(2,3,4,5,6);
(2,3,4,5,6,1)

def reverse=(list1 [] pofx i);

reverse(1,2,3,4,5);
(5,4,3,2,1)

def rec list2 a=[Δg[Δf[Δl[null l?a!
                    list2(g(f(h l))a)g f (t l)!]]]];

list2 [] pfx sq(1,2,3,4,5);
(25,16,9,4,1)
list2 0 p1 i(1,2,3,4,5);
15

list2 [] pofx i(1,2,3);
(1,2,3)

list1 [] pfx i(1,2,3);
(1,2,3)

List2 is a recursion schema similar to list1.  We

can show that list1 a g f x = list2 a g f(reverse x)
```

4. List Structures

A recursion schema for list structures i.e. a structure defined as either being atomic or being a list structure-list is

```
def rec ls a=[Δg[Δf[Δl[/lst l?listl a g(ls a g f)l!
                f l!]]]];
def lstr=(1,(2,(3,4,5),6),7,8):
lstr;
(1,(2,(3,4,5),6),7,8)
```

/

This uses a built in predicate /lst which tests whether its argument is a list.

Some of the functions defined for lists can be extended to be applicable to list structures. e.g.,

```
ls 0 pl i lstr;
36

ls 1 mt i lstr;
40320

def mapls f=[ls []pfx f];

mapls sq lstr;
(1,(4,(9,16,25),36),49,64)

ls [] pofx i lstr;
(8,7,(6,(5,4,3),2),1)

listl [] pofx i lstr;
(8,7,(2,(3,4,5),6),1)
```

Recursion schemata for functions which operate on two lists of equal length or two list structures having equal structure follow.

```
def rec zip a=[Δg[Δf[Δx[Δy[null x?a!g(f(h x)(h y))
          (zip a g f(t x)(t y))!]]]]];
```

The function f operates on two corresponding elements and g on the result and the result of applying the function to the tails of the two lists.

```
def zips = (zip [] pfx pair);
def pair x=[Δy[x,y]];
def nums=(1,2,3,4,5,6,7);
def zips=(zips [] pfx pair);
```

```
zips nums nums;
((1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7))

def sprod=(zip 0 pl mt);

sprod(1,2,3)(4,5,6);
32

def rec zipls a=[Δg[Δf[Δx[Δy[/lst x?zip a
                g(zipls a g f)x y!f x y!]]]]];

zipls [] pfx pair lstr lstr;
((1,1),((2,2),((3,3),(4,4),(5,5)),(6,6)),(7,7),(8,8))

zipls 0 pl mt lstr lstr;
204
```

5. Sets and relations

```
def true=(0=0);

def false=(0=1);

def a x=[Δy[x?y!false!]];

def o x=[Δy[x?true!y!]];

def n x=[x?false!true!];
```

These definitions introduce the truth values true and false and the functions and (a), or (o) and not (n). The following functions are analogues of functions on sets in which a list represents a set.

```
def exists p=[listl false o p];

exists(eq 5)(2,4,5,6,7)[tf];
T

def all p=[listl true a p];

def odd n=[n-(n/2*2)=1];

all odd(1,2,3,5,7)[tf];
F
all odd(1,3,5,7)[tf];
T

def filter p=[listl [] i Δx[p x?pfx x! i !]];

filter odd(1,2,3,4,5,6):
(1,3,5)
```

The function exists tests whether a member of the list has property p, the function all tests whether all members have property p, filter selects from a list all those having property p and belongs tests whether x is a member of the list l. The function incl will test whether y is included in x and eqset tests whether two sets are equal.

```
def belongs l=[Δx[exists(eq x)l]];

belongs(1,2,3,4,5)3[tf];
T

def incl x=[Δy[all(belongs x)y]];

incl(1,2,3,4,5)(4,2,3)[tf];
T

def eqset x=[Δy[a(incl x y)(incl y x)]];

eqset(1,2,3)(3,1,2)[tf];
T

def intsn=(b filter belongs);

intsn(1,2,3,4,5)(2,3,4,5,6,7,8);
(2,3,4,5)
```

```
def diff x=[Δy[filter(b n(belongs y))x]];

diff(1,2,3,4,5)(1,3,5,7,9);
(2,4)

def union x=[Δy[app(diff x y)y]];

union(1,3,5,7)(2,4,6,8);
(1,3,5,7,2,4,6,8)

def rec eqst x=[Δy[null x?null y!h x=
     h y?eqst(t x)(t y)!false!!]];
```

In the following relations will be treated as mappings from an individual to a set (list). Ad hoc relations may be constructed as follows.

```
def rel x=[Δy[Δz[eqst x z?y![]!]]];

rel joe(u jim)joe;
(JIM)
```

Functions which operate on and produce new relations can be defined as follows:

```
def diff x=[Δy[filter(b n Δz[exists(eqst z)y])x]];

diff([ds],[gs])([ds],[lk]);
(GS)

def union x=[Δy[app(diff x y)y]];

union([as],[df])([re],[uy]);
(AS,DF,RE,UY)

def orr=(phi union);

def brother=(rel joe(al,bob,clarence));

def sister=(rel joe(anne,babs));

orr brother sister joe;
(AL,BOB,CLARENCE,ANNE,BABS)

def sumset=(listl [] union i);

sumset(([qw],[er]),([qw],[kj]),([jh],[mn])):
(ER,QW,KJ,JH,MN)

def nullr x=[[]];

def relist=(zip nullr orr rel);

def father=(relist(al,don,claude)(u don, u ed, u
     fred));

father don;
(ED)

def prodf f=[Δg[Δx[sumset(map g(f x))]]];

def grandfather=(prodf father father);

grandfather al;
(ED)

def brother=(relist(fred,al)((jim,joe),(tom,harry)));

def son=(rel don(al,ian));

brother fred;
(JIM,JOE)

def self=u;

def butnot=(phi diff);

def brother=(orr brother(butnot(prodf father son)
          self));

brother al;
(TOM,HARRY,IAN)

def trcl r=[rec tc=(orr r(prodf r tc)) tc];

trcl father al;
(DON,ED)

def map2 f=[Δx[Δy[ccat(map(Δz[map(f z)y])x)]]];
```

If we regard a list as a set then map f produces the

```
set [f x | x ∈ A] and map2 f produces the set

[f x y | x ∈ A, y ∈ B]

def xprod=(map2 pair);

xprod(1,2,3,4)(5,6,7,8);
((1,5),(1,6),(1,7),(1,8),(2,5),(2,6),(2,7),(2,8),
(3,5),(3,6),(3,7),(3,8),(4,5),(4,6),(4,7),(4,8))

xprod(1,2,3)(4,5,6);
((1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),
(3,6))

map2 p1(1,2,3)(4,5,6);
(5,6,7,6,7,8,7,8,9)

def mmult x=[Δy[map Δz[map(sp z)x]y]];

mmult((1,2),(3,4))((4,5)(6,7));
((14,32),(20,46))

The function for matrix multiplication has a similar

structure to map2 but the list structure is retained.

A matrix is represented by a list of lists.

def cxp=(zip [] pfx p1);

def cxm(x,y)=[Δ(s,t)[x*s-y*t,x*t+y*s]];

def cxsp=(zip(0,0)cxp cxm);

cxp(1,2)(3,4);
(4,6)

cxm(1,2)(3,4)
(-5,10)

cxsp((1,2),(3,4))((5,6),(7,8));
(-18,68)

def cxmmult x=[Δy[map Δz[map(cxsp z)x]y]];

cxmmult(((1,2),(3,4)),((5,6),(7,8)))(((9,10),
(11,12)),((13,14),(15,16)));
(((-26,108),(-34,276)),((-34,148),(-42,380)))

     The multiplication of matrices of complex

numbers uses the same format but p1 is replaced

by cxp (complex plus) and mt by cxm and 0 by (0,0).

def rec trp m=
[null(h m)?[]!pfx(map h m)(trp(map t m)))!];

trp((1,2),(3,4));
((1,3),(2,4))
```

6. Syntactic analysis

Functions for the analysis of the syntax of strings can be constructed in a systematic way by defining a set of elementary analysers which we will call 'parsers' and then by defining functions from parsers to parsers. Suppose a parser operates on a string to produce a list of three items. The first is a truth value, if true the parser has found what it is looking for at the head of the string, if false it has not. The second item is something produced from the string, and the third is a character string which is the original string if the first item is false, otherwise it is the string with the recognized part removed.

```
def char s=[null s?false,[],s!true,h s,t s!];

def pr(x,y,z)=[x([true],[false]),y,z];

pr(char[qwert]);
(TRUE,Q,WERT)

pr(char[]);
(FALSE,().())

def q p=[Δs[let(b1,c1,s1)=(char s) b1(p c1)?true,c1,
     s1!false,[],s!]];
```

Char tests whether the string is null and q operates on a predicate to produce a parser.

```
pr(q letter[abcd]);
(TRUE,A,BCD)

pr(q digit[abcd]);
(FALSE,(),ABCD)

def eqch=(b q eq);

pr(eqch'a[abcd]);
(TRUE,A,BCD)

pr(eqch'a[bcde]);
(FALSE,(),BCDE)
```

The function eqch operates on a character and produces a parser.

```
def un f=[Δg[Δs[let(b1,c1,s1)=(f s)b1?b1,c1,s1!
     let(b2,c2,s2)=(g s)b2?b2,c2,s2!false,[],s!!]]];
```

The function un operates on two parses to produce a third which first tests f applicable and if not tests if g is applicable.

```
pr(un(eqch'a)(eqch'b)[abcd]);
(TRUE,A,BCD)

def ab=(un(eqch'a)(eqch'b));

pr(ab[abcd]);
(TRUE,A,BCD)

pr(ab[bcd]);
(TRUE,B,CD)

pr(ab[cdef]);
(FALSE,(),CDEF)

def cc h=[Δf[Δg[Δs[let(b1,c1,s1)=
     (f s)b1?let(b2,c2,s2)=(g s1)
b2?b2,h c1 c2,s2!false,[],s!!false,[],s!]]]];
```

The function cc operates on two parsers to produce a parser which recognizes an f followed by a g. If it recognizes both then the results are combined by a f[illegible]n.

```
def ct x=[Δy!x,y]];

def cab=(cc ct(eqch'a)(eqch'b));

pr(cab[abcd]);
(TRUE,AB,CD)

pr(cab[acde]);
(FALSE,(),ACDE)

pr(cab[qwe]);
(FALSE,(),QWE)

def rec qufyl h=[Δg[Δc[Δs[let(b1,c1,s1)=(g s)

b1?qufyl h g(h c c1)s1!true,c,s!]]]];

pr(qufyl ct ab 'a[aababakjhgfd]);
(TRUE,(((((AA,A),B),A),B),A),KJHGFD)

def qufy h=[Δf[Δg[Δs[let(b1,c1,s1)=(f s)

b1?qufyl h g c1 s!false,[],s!]]]];
```

The function qufyl recognises any number of g's and combines their results with an initial value c using the function h.

The function qufy recognises a phrase which starts with an f and is followed by any number of g's. In fact it looks for the longest sequence of g's which it can find.

```
def cat x=[Δy[x y]];

def iden=(qufy cat(q letter)(un(q letter)(q digit)));

pr(iden[asf6,bcvx]);
(TRUE,ASF6,,BCVX)

pr(iden[apple)rrrrr]);
(TRUE,APPLE,)RRRRR)

pr(iden[a4f*uuu]):
(TRUE,A4F,*UUU)

pr(iden[)))))))]);
(FALSE,(),)))))))

def cnum x=[Δy!10*x+y]];

def int=(qufy cnum(q digit)(q digit));

pr(int[123qwe]);
(TRUE,123,QWE)

pr(int[5yty]);
(TRUE,5,YTY)

def edit f=[Δg[Δs[let(b1,c1,s1)=(g s)

b1?b1,f c1,s1!false,[],s!]]];

pr(edit(b sq \chin)(q digit)[4tredf]);
(TRUE,16,TREDF)

def none s=[false,[],s];

def empty s=[true,[],s];

def unlist=(list1 none un i);

pr(unlist(map eqch[abcd])[asdfgh]);
(TRUE,A,SDFGH)

def seek l=[unlist(map eqch l)];

pr(seek[abcd][qwer]);
(FALSE,(),QWER)

def cclist=(list1 empty(cc pfx)i);

def eqst l=[cclist(map eqch l)];

pr(eqst[abcd][abcdefgh]);
(TRUE,ABCD,EFGH)

pr(eqst[abcd][abcfrg]);
(FALSE,(),ABCFRG)

def unmap f=[Δg[Δs[let(b1,c1,s1)=
     (g s)b1?f c1 s1!false,[],s!]]];
```

```
def precinops=(relist[+-*/><=]
     ([*/],[*/],[],[],[-+*/],[-+*/],[-+*/]);
```

This is a precedence relation for infixed operators. When applied to an operator character it produces a list of those which have stronger precedence.

```
def rec scinops o=[let quop op=[Δs[let(b1,c1,s1)=
     (scinops(precinops op)s)
b1?b1,cat c1 op,s1!false,[],s!]]qufy cat char(unmap
          quop(unlist(map eqst o)))];

def mdpm=([*],[/],[+],[-]);

pr(scinops mdpm[a+b*c-d]);
(TRUE,ABC*+D-,())

pr(scinops mdpm[a+b<c-d]);
(TRUE,AB+,<C-D)

def mdpmgle=(cat mdpm([>],[<],[=]));

mdpmgle;
(*,/,+,-,>,<,=)

pr(scinops mdpmgle[a+b>c*d]);
(TRUE,AB+CD*>,())
```

The function scinops translates an expression to reverse Polish. The precedence relation is specified by precinops, and its argument is the list of infixed operators permitted.

7. Sequences

```
def first(x,y)=[x];

def second(x,y)=[y];

def hs f=[first(f[])];

def ts f=[second(f[])];

def rec gen f=[Δx[Δy[x,gen f(f x)]]];

def ths n=[Δf[hs(z n ts f)]];
```

The function gen f x represents the sequence x,f x,f(f x),f(f(f x))...etc. The function ths n selects the nth member of the sequence.

```
def suc x=[x+1];

ths 5(gen suc 0);
5
```

The sequence may be transformed to another sequence by functions such as maps and filters defined as follows.

```
def rec maps g=[Δf[Δy[g(hs f),maps g(ts f)]]];

def integer=(gen suc 0);

ths 5(maps sq integer);
25

def rec filters p=[Δf[p(hs f)?Δy[hs f,filters
     p(ts f)]!filters p(ts f)!]];

ths 5(filters odd integer);
11
```

```
def rec mus p=[Δf[p(hs f)?hs f!mus p(ts f)!]];

mus(gr 10)integer;
11

mus p f is the first member of the sequence f hav-

ing property p.

def rec while p=[Δf[p(hs f)?pfx(hs f)(while
                    p(ts f))![]!]];

def odds=(filters odd integer);

def less x=[Δy[y<x]];

while(less 15)odds;
(1,3,5,7,9,11,13)

def div n=[Δx[n-(n/x)*x=0]];

def prime x=[let small y=
     [n(2*y>x)]n(exists(div x)(while small
                (gen suc 2)))];

def primes=[filters prime(gen suc 2));

ths 7 primes;
19

while(less 12)primes;
(2,3,5,7,11)
```

8. Abstract Syntax

```
def zeroto n=[while(less(n+1))integer];

zeroto 6;
(0,1,2,3,4,5,6)

def(any,int,ch,lstg,bf,cl,dun,cpr)=(zeroto 7);

def mk=pfx;

def sels l=[map th(zeroto(lth l-1))];

def is x=(x=any?k true!x=int?\int!x=
     ch?\ch!x=lstg?\lst!x=bf?\bf!x=
          cl?\cl!Δy[x=h y]!!!!!!!];

def preds l=[map is(t l)];

def du=(mk d un); def cp =(mkcpr)i

def parts =t;
```

We can, by using du and cp, create a representation of a set of data structures. By applying parts, sets, preds to the representation we can create functions for analysing members of the set and mk is used for creating members of the set. The functions defined above take from the user the burden of specifying how the information is represented inside the machine, he has only to specify its logical structure. An example follows.

```
def rec intree=(du(int,cp(intree,intree)));
```

The data structure called intree has two formats. It is either an integer or is a pair of intrees.

```
def(atomc,nonatomc)=(parts intree);

def(atom,nonatom)=(preds intree);
```

```
def(left,right)=(sels nonatomc):

def ctree s=[Δy[mk nonatomc (x,y)]]:
```

The functions just defined can now be used to create trees e.g.

```
def rec ltot l=[null(t l)?h l!ctree(h l)(ltot(t l))!];
```

A recursion schema for analysing trees can be defined as follows.

```
def rec tree g=[Δf[Δx[atom x?f x!
g(tree g f(left x))(tree g f(right x))!]]];

ptr(ltot(1,2,3,4,5));
(1:(2:(3:(4:5))))

def l=(ltot(1,2,3,4,5));

tree pl i l;
15

def revt=(tree(c ctree)i);

ptr(revt l);
((((5:4):3):2):1)

def mapt f=[tree ctree f];

ptr(mapt sq l);
(1:(4:(9:(16:25))))

def subt x=[Δy[tree ctree Δz[z=x?y!z!]]];

ptr(sub 4 1 1);
(1:(2:(3:((1:(2:(3:(4:5))))):5))))

ptr(ctree 1 l);
((1:(2:(3:(4:5)))):(1:(2:(3:(4:5)))))
```

9. Implementation

The syntax of MCG will be described as a set of character strings. Some of the examples given demonstrate that there are certain phrases which are not really necessary but are introduced only for programming convenience. It is possible to systematically replace certain phrases by others and reduce the language to a subset of itself.

MCG is implemented as a two stage process. The first stage produces a list of instructions and the second interprets this list. These two functions will be described in MCG language.

9.1 Syntax

The syntax will be described in terms of certain basic sets of character strings, namely integer, character, quote, bra, ket, sqbra, sqket, space, delta, let, rec, equals, comma, question, exclam, empty, identifier, together with functions from sets to sets: union, ccxp, unlist, ccxplist, sqbracket, bracket, diff, qualify defined as follows:

union and diff have been defined in the examples

```
def ccxp=(map2 app);
def unlist=(list1 [] union i);
def ccxplist=(list1 [] ccxp i);
def sqbracket x=[ccxplist(sqbra,x,sqket)];
def bracket x=[ccxplist(bra,x,ket)];
def rec qualify x=[Δy[union x(ccxp(qualify x y)y)]];
def spaces=(qualify empty space);
```

9.1.1 Literals

```
def literal=(unlist(integer,
                    sqbracket quoteablestring,
                    ccxp quote character));

def rec quoteablestring=
     (unlist(sqbracket quoteablestring,
     ccxp quote character,
     diff character(unlist(sqbra,sqket,quote)),
     ccxp quoteablestring quoteablestring,
     empty));
```

A quoted string may be replaced by quoted characters separated by commas. The value of a literal is either an integer, a character or a character-list.

9.1.2 Bound variable

```
def rec bv=
  (union identifier(bracket(qualify
                        bv(ccxp comma bv))));
```

A list structure of identifiers is produced from a bv.

9.1.3 Primary

```
def rec primary=
     ( unlist (ccxp spaces primary,
               identifier,
               literal,
               bracket expression,
                ccxplist(delta,bv,sqbracket expression).
                qualifiedexpression));

def qualifiedexpression=(ccxp defn expression);

def defn=(ccxp(union let rec)(union(ccxplist(bv,
                                  equals, primary))

(ccxplist(identifier,bv,equals,
                    sqbracket expression))):
```

The functional definition can be reduced as follows.

```
ccxplist(union let rec,identifier,bv,equals,
                              sqbracket expression)
```

can be replaced by

```
ccxplist(union let rec,identifier, equals,delta,bv,
                    sqbracket expression)
```

The rec option can be reduced to a let phrase as follows:

```
ccxplist(rec,bv,equals,primary)
```

can be replaced by

```
ccxplist(let,bv,equals,why,delta,bv,primary)
```

where why is the character string set for the name of the function y given in the example.

9.1.4 Combinations

```
def combination=(qualify primary primary);
```

A combination denotes application of function to argument. The syntax of a combination can be simplified by using a fully bracketed notation. i.e.

```
bracket(ccxp primary primary)
```

9.1.5 Infixed Operator Expressions

```
def inopexp=(scinops([+],[-],[*],[/],[<],[>],[=]));

def scinops x=[let quop(op)=
     (ccxp(u op)(scinops(precinops op)))
          qualify combination(union(map quop x))];
```

The infixed operator expressions can be transformed to combinations by using prefixed operators instead.

9.1.6 Expressions

```
def expression=(union(qualify inopexp(ccxp comma
                                      inopexp)) ·

(ccxplist(expression,question,expression exclam,
                expression,exclam)));
```

The phrases for lists whose items are separated by commas can be replaced by combinations involving the function pfx and the null list. The conditional expression can be replaced as follows.

```
ccxplist(expression1,question,
     expression2,exclam,
     expression3,exclam);
```

can be replaced by

```
ccxplist(expression1,
     bracket(ccxplist(delta,nullist,expression2,
          comma,delta,nullist,expression3)),
          nullist)
```

The two arms are converted to expressions denoting functions applicable to the nullist, and paired. The condition selects one function and applies it to a nullist argument. This device ensures that evaluation of the arms is delayed until one has been selected.

If these equivalences are employed then the syntax is reduced to that given below. In average

the constants y, pfx and nullist have been introduced.

```
def primary=(unlist(ccxp spaces primary,
                    identifier,
                    integer,
                    ccxp quote character,
                    ccxplist(delta,bv,
                         sqbracket primary),
                    bracket(ccxp primary primary));

def bv=(union(identifier,
          bracket(qualify bv(ccxp comma bv))));
```


9.2 Abstract Syntax

The abstract syntax of a primary is defined as follows:

```
def list x=[rec l=du(cp (), cp(x,1))1];
```

The predicate null, selectors h & t and constrcter pfx will be used for lists.

```
def identifier=(list character);

def liststructure x=[rec l=(du(x,list l)) l];

def bv=(liststructure identifier);

def(atom,nonatom)=(preds bv);

def rec primary=(du(identifier,
                    integer,
                    character,
                    cp(bv,primary),
                    cp(primary,primary)));

def(idenc,intc,chc,lamexc,cbnc)=(parts primary);

def(id,int,ch,lam,cbn)=(preds primary);

def(bdv,body)=(sels lamexc);
def(rator,rand)=(sels cbnc);
```

The system is implemented in two phases. The first phase compiles a primary to a list of instructions. The second phase interprets these instructions and produces the value of an expression. The abstract syntax of instructions is defined as follows:

```
def instruction=(du(cp(u any),
                    cp(list integer),
                    cp(list instruction),
                    cp () ));

def(lobc,lopc,locc,ap)=(parts instruction);

def(lob,lop,loc,ap)=(preds instruction);

def obj=(h(sels lobc));

def posn=(h(sels lopc));

def subr=(h(sels locc));
```

9.3 Compiler

The compiler is defined as follows. It operates on a list of named objects E and an identifier-list F and an expression x and produces a list of instructions.

```
def rec compile E=
[ΔF[Δx[

id x? let(b,p)=posnlst F x
     b? u(mk lopc(u p))!
        u(mk lobc(u(lookup E x)))!

or(int x)(ch x)? u(mk lobc(u x))!

lam x?  u(mk locc(compile E(pfx(bv x)F)(body x)))!

cbn x?  ccat(compile E F(rand x),
             compile E F(rator x),
             u(mk apc () ))!!!!]]];
```

If the expression is an identifier then the list of identifier list structures F is first searched for it. If the identifier is found in F then its position in the form of a list of integers is made into a lop type of instruction. If the identifier is not found in F then the list E is searched for it. If it is found in E then the corresponding object is made into a lob type of instruction. If the identifier is not found in E then its value is taken to be the list of characters which make up the identifier.

If the expression is a lambda expression, its

body is compiled in a new F environment which has its bound variable prefixed to the old F and the result is made into a loc type instruction.

If the expression is a combination then its operator and operand are both compiled and the results are concatenated with a trailing ap type of instruction.

The two functions for searching are defined as follows:

```
def rec lookup E=
[Δx[null E?x!
     h(h E)=x? snd(h E)!
               lookup(t E)x!!]];

def rec(posn1,posn1s)=
([Δ1[Δn[Δx[null 1?false,[]!
               let(b1,p1)=(posn1s(h 1)0 x)
             b1?true,pfx n p1!
               posn1(t 1)(n+1)x]]]],
[Δ1[Δn[Δx[atom 1?eqst x 1?true,[]!
               false,[]!!
          posn1 1 n x!]]]]);

def posn1st 1=[Δx[posn1 1 0 x]]
```

9.4 Interpeter

The interpeter executes programs and operates on a four component state.

1) The first component S is a list of objects used in a last in first out manner for working space.

2) The second component E is a list of objects used for storing arguments of functions.

3) The third component C is a list of instructions.

4) The fourth component D is used for storing a state upon 'subroutine' entry.

```
def rec interpret(S,E,C,D)=
[null C?null D?S,E,C,D!
               let(S1,E1,C1,D1)=D
               hS:S1,E1,C1,D1!!
          interpret(tr(h C)(S,E,t C,D))!]
     where

def tr x=[Δ(S,E,C,D)[
     lob x?pfx(obj x)S,E,C,D!
     lop x?pfx(get E(posn x))S,E,C,D!
     loc x?pfx(cclosure(subr x,E)S,E,C,D!
     ap x?
      bf(h S)?pfx((h S)(snd S))(t(t S)),E,C,D!
     closure(h S)? (),pfx(snd S)(env(h S)),prog(h S),
          (t(t S),E,C,D)!!!!!!]]

def rec get 1=[Δp[null p?1!get(th(h p)1)(t p)!]];

     where

def function=du(cp(u bf),cp(list instruction,
                                 list any))
```

```
def(bf,closure)=(preds function);
def(bfc,closurec)=(parts function);
def(prog,env)=(sels closurec);
def cclosure=(mk closurec);
```

If the next instruction (x) is a lob then the object associated with it is prefixed to the S component.

If x is a lop instruction then an object is selected from E using the list of integers and prefixed to S.

If x is a loc then a closure is prefixed to S. This closure is constructed from the instruction list attached to the instruction and the current E.

If x is an ap then if the head of S is a basic function then it is applied to the second of S and the result is prefixed to the doubly beheaded S. If the head of S is a closure then the state except for the closure and its argument is stored in the D component and the new S is the nullist, the new E is formed by prefixing the argument to the environment part of the closure, and the new C is formed from the program part of the closure.

The function interpret repeatedly applies tr to the first instruction and the state until the C component is the nullist. In this case if the D component is not null then the state stored in the D component is restored and the head of the old S is prefixed to the new S. If the D component is null then the result is the original state.

The whole system can now be described as follows. Suppose there is a parser for messages called scan-message which produces a triple whose second member is a message defined as follows.

```
def message=du(cp(identifier,primary),
               cp(u primary));
def(defnc,expc)=(parts message);
def(defn,exp)=(preds message);
def(nee,niens)=(sels defnc);
```

9.5 MCG System

The complete Mcg system can now be described in terms of a function called 'mcg' which operates on 1) I (the input) which is a character-list, 2) O (the output) which is a character-list and 3) E an environment which is an identifier object pair list.

```
def value E=[Δx[
let(S,E,C,D)=(interpret([],[],compile E[]x,[]))
              h S]]

def rec mcg(i,o,e)=
[let(b1,c1,s1)=(message i)
     b1?defn c1?mcg(s1,o,pfx(nee c1,
                    value e(niens c1))e)!
     mcg(s1,app(value e c1)o,e)!!
     mcg(movetonextmessage i,o,e)!]
```

If the message is a definition its definiens is evaluated in E, paired with its identifier and prefixed to E. The output is unchanged. If the message is an expression then its value is appended to the output and E is unchanged. In both cases the input is changed to the character string remaining after the message has been taken off.

Acknowledgements

The first section is a fairly standard method of generating partial recursive functions in terms of the lambda notation. The second is an extension of the techniques to functions which operate on lists and list structures. The section on sets was suggested by a paper by Burstall (2). The parsing functions in section 6 have been used to write a compiler from the Mcg language to assembly code. The functions which represent sequences are a functional analogue to co-routines (3). The abstract syntax functions are due to McCarthy (5). The technique of employing recursion schemata can be extended to produce functions on any data structure defined in this way. The method of describing the concrete syntax is similar to Backus Normal Form (1) but is extended to permit extra functions from sets of character strings to sets of character strings. The implementations are based upon Landin's SECD machine (4), and the system is similar to Strachey's GPM (6). In fact Mcg is an abbreviation for "macrogenerator".

A first version of Mcg was implemented by Carol A. Conn and was purely interpretive. A second version, whose essentials are described here, was programmed by Paul O. Backer.

References

(1) Backus, J. W. The Syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference presented at ICIP Paris 1959.

(2) Burstall, R.M. A combinatory approach to relational question answering and syntax analysis. Nato Summer School, Copenhagen, August 1967.

(3) Conway, M. E. Design of a seperable transition-diagram compiler CACM 6 7 (1963) 396-408

(4) Landin, P. J. The mechanical evaluation of expressions. Comput. J. 6 4 (Jan 1964) 308-320

5) McCarthy J. A basis for a mathematical theory, of computation. Computer Programming and Formal Systems, P. Braffort and D. Hirshberg (Eds) North Holland, Amsterdam 1963

(6) Strachey, C. A general purpose macrogenerator. Comput. J. 8 3 (Oct. 1965) 225-241.

5705-2
121