W. H. Burge

Combinatory Programming and Combinatorial Analysis

Abstract: The principal purpose of this paper is to illustrate by means of simple examples a technique for deriving programs and generating functions from set descriptions. The paper discusses certain interesting correspondences among types of trees, which follow from the use of the technique, and it demonstrates close connections between programming techniques and some aspects of combinatorial theory.

Introduction

This paper introduces methods of simplifying programming, which, although not entirely new, have not been used in practice to any great extent, perhaps because they require certain features that many current programming languages do not provide. Instead of the usual method of producing programs by the step-by-step sequencing of instructions, programs are produced by the application of a general-purpose function to its arguments. Such programs are easier to understand and their properties are more easily discovered and proved. The method involves certain ways of omitting irrelevant details that encumber programs, although the extra details that need to be supplied are precisely specified. By omitting the unnecessary details, it is possible to reveal the essential structure of the algorithm so that is can be more easily communicated and analyzed. This abbreviation, together with certain program constructions, facilitates the expression of more complex algorithms than is presently feasible.

A program has to be studied throughout all the stages of its preparation. In the beginning, it has to be examined to see whether it appears to be correct. Further studies might include a proof that it has a certain property or an analysis of the number of times each part of the program will be executed. These last two deeper studies of programs naturally should give the programmer a greater conviction of their correctness. For the rather limited types of programs considered in this paper, certain properties can be proved in an almost mechanical fashion, and an analysis of the running time of a program can easily be accomplished by constructing a generating function that has the same structure as the program itself.

Under the technique for constructing a program described in this paper, the first step is to write a general-purpose program that is a mere skeleton. The second step is to supply arguments that specify the actions to be taken within the skeleton. The skeletal program can be considered to typify the whole family of programs obtained in this way, and any properties or timing analyses of the general-purpose program are then applicable to all the members of this family.

The programs that have been chosen to illustrate the technique in this paper all operate on rooted, ordered trees. The general-purpose skeleton embodies the method of scanning the structure, and the arguments supplied specify the actions to be taken during the scanning.

The general-purpose programs can be produced in a mechanical fashion from the description of the set of trees being scanned. The set of trees considered have components that are either *atomic*, meaning their internal structure is not examined by the general-purpose program, or *composite*, and their internal structure is examined. An enumerating generating function that counts the number of different structures having n atomic components can also be constructed automatically from the description of the set of trees.

A programmer can describe a set of information structures without specifying the nature of its atomic components. It is possible, for example, to define a set of

lists and to write functions on the lists without specifying the nature of the list elements, which are treated as atomic as far as the general-purpose list function is concerned. A function that is concerned with the nature of the list elements can be supplied as an argument to the list function. An element of a list may itself have internal structure; for example, it may be a tree, containing atomic components of its own. It is therefore possible to create a general-purpose function for a list of trees from the corresponding functions for lists and for trees by supplying one as an argument to the other. In a similar way, the generating function for counting the number of lists of trees with n atomic components can be obtained by substituting the generating function of the trees for the appropriate variable in the generating function of the lists.

The generating function classifies the members of a set of structures according to the number of atomic components. Each set of structures with n atomic components can itself be classified according to another criterion, and this two-way classification is represented by a generating function in which the coefficients of powers of one variable are generating functions in another variable. One type of problem that often arises in the analysis of algorithms is that of finding the sum of the distances from the root of a tree to each atomic component. This sum is often called the *weight* of the tree. The passage from one level of the tree to the next stands for the performance of some operation, and the weight of the tree divided by the number of atomic components is the expected number of operations needed to reach an atomic component of the tree. This type of problem can be solved by using a generating function that classifies the structures both by number of nodes and by weight, and the method for deriving this generating function almost immediately from the structure description of the set is given.

Data structures

The method for describing new sets of data structures was introduced for use in programming languages by McCarthy [1] and has been used to specify the semantics of programming languages (e.g., Algol 60 [2] and PL/1 [3]) and occurs as part of the Algol 68 programming language [4]. The description of a set is often referred to as its *abstract syntax*, and although there might be notational differences in the references above, the main ideas are much the same.

We use the two functions *Cartesian product* and *direct union* to produce a set from two or more sets, and use a notation for naming the predicates, selectors, and constructors for members of the sets produced.

The Cartesian product $A \times B$ of two sets A and B is the set of ordered pairs (a,b) where $a \in A$ and $b \in B$. This product is extended to operate on any number of sets. Instead of $A \times B \times C$, we use cp(A,B,C) to denote a set of structures each member of which has three components whose first is of type A, whose second is of type B, and whose third is of type C. This change allows one-component or zero-component structures to be defined, and these will be written cp(A) and cp(), respectively.

Functions for selecting the components (*selectors*) are needed and are assumed to be both created and named by a definition or declaration. Such definitions are preceded by the word def to emphasize that a definition follows. For example, the following defines the selectors *first*, *second*, and *third*, which are all applicable to cp(A,B,C):

def *first*,*second*,*third* = *selectors*(cp(A,B,C)).

Alternatively the set could first be named as follows:

def *3-tuple* = cp(A,B,C)

def *first*,*second*,*third* = *selectors 3-tuple*.

The same information could have been conveyed by the following English sentence [2]:

A 3-tuple has a first, which is an A,
and a second, which is a B,
and a third, which is a C.

In Algol 68 style the same definition could be written:

mode 3-tuple = struct (A *first*, B *second*, C *third*).

In all cases the definition is assumed to create functions of the following types

first $\in$ (*3-tuple* $\rightarrow A$)

second $\in$ (*3-tuple* $\rightarrow B$)

third $\in$ (*3-tuple* $\rightarrow C$),

in which $A \rightarrow B$ denotes the set of functions from A to B.

A program fragment that operates on a 3-tuple takes the form

$F(FA(first\ (x)), FB(second\ (x)), FC(third\ (x)))$,

in which x is a 3-tuple, and the functions FA, FB, and FC are applied to the components producing results that are combined using a function F.

A function that is applicable to more than one argument may be changed to a single-argument function by using a notational device of combinatory logic [5] in which the expression $(f\ x)y$ denotes the application of a function f to x and y in two stages, so to speak. The function f is first applied to x producing a function that is then applied to y. This implicit application operator is assumed to associate to the left, so that $f\ x\ y$ means the same thing as $(f\ x)y$. Note that parentheses are never needed to enclose a sole identifier. All functions are assumed to have one argument; in the case of f in an expression like $f(x,y)$,

it is still treated as a function of one argument, which in this case is an ordered pair.

Functions (called *constructors*) are also needed for constructing structured objects from a list of their components and are defined according to the following style:

def *triple x y z* = *construct 3-tuple* (x,y,z) .

The constructing and selecting functions are related as follows:

triple (*first u*)(*second u*)(*third u*) = *u*

first (*triple x y z*) = *x*

second (*triple x y z*) = *y*

third (*triple x y z*) = *z*

in which u is a 3-tuple and x, y, and z are members of A, B, and C, respectively.

The other function that is employed to create new sets is the direct union, written du(A,B), and is used when members of a set can have the two different formats A or B. In order to write programs, it is necessary to be able to test whether a member of du(A,B) is an A or a B. These predicates and their names are defined in the following style:

def *is-A*, *is-B*, *is-C* = *predicates*(du(A,B,C)) .

The associated axioms merely state that the predicates are effective and take the form:

is-A (x) = true if $x \in A$
false otherwise.

In practice this means that some additional information has to be added to make it possible to recognize members of A, B, and C when they appear as members of du(A,B,C) .

The piece of program that naturally corresponds to du(A,B,C) is

```
if is-A (x)
then FA (x)
else if is-B (x)
    then FB (x)
    else FC (x)
```

in which tests are first made for the alternative formats and then a function of the correct type is applied.

An inverse to du, called *parts*, is often useful for separating and naming the alternatives:

parts(du(A,B,C,D)) = A,B,C,D .

The number of members of cp(A,B) or du(A,B) is the product or sum of the number of members of A and B. It is possible to provide more information about these sets as follows. Each object under consideration either has components or is atomic and has no internal structure. The members of each set can be classified by the number of atomic components they contain. This classification can be expressed as an enumerating generating function in which the coefficient of x^n is the number of members of the set having n atomic components. Suppose that $A(x)$ and $B(x)$ are two generating functions of this sort, thus:

$$A(x) = \sum_{n=0}^{\infty} a_n x^n, \; B(x) = \sum_{n=0}^{\infty} b_n x^n ,$$

in which a_n is the number of members of A having exactly n atomic components, and b_n is the number of members of B having n atomic components. Then the number of members of cp(A,B) having exactly n atomic components is the coefficient of x in the product of $A(x)$ and $B(x)$, i.e.:

$$A(x)B(x) = \sum_{n=0}^{\infty} \sum_{k=0}^{n} a_k b_{n-k} x^n .$$

In this sum the term $a_k b_{n-k}$ is the number of members of cp(A,B) in which the first of the pair has k atomic components and the second has $n-k$.

The number of members of du(A,B) having n atomic components is $a_n + b_n$. The generating function for du(A,B) is therefore found by adding the generating functions for A and B,

$$A(x) + B(x) = \sum_{n=0}^{\infty} (a_n + b_n)x .$$

The same correspondences hold between sets of trees having more than one type of atomic component and multivariable generating functions. The generating function of a Cartesian product is the product of the generating functions of the component sets, and the generating function of a direct union is the sum of the generating functions for the alternative sets.

The three-way correspondence among structure descriptions, programs, and generating functions is summarized in the table below.

Sets	*Program*	*Generating function*
atomic A	f, a function on atomic components	x, a variable
cp(A,B)	$F(FA(first(x)), FB(second(x))))$	$A(x) \cdot B(x)$
du(A,B)	if *is-A* then $FA(x)$ else $FB(x)$	$A(x) + B(x)$
cp()	a, a variable	$x^0 = 1$
$A = ..B..B.$	$FA(FB) = ..FB..FB.$	$A(B(x))$

In the last line $A = ..B..B.$ means a set A is defined in terms of a set B, $FA(FB)$ means that the function FA

is defined in terms of the function FB, and $A(B(x))$ means that $B(x)$ is substituted for x in $A(x)$. A number of examples of these correspondences are given next.

Lists

Lists are the most commonly used data structure in programming, and the definition of a list of elements of type A is:

An *A-list*
  is either null
    or has a head, which is an A,
      and a tail, which is an *A-list*,

or

<u>def</u> *list A* = du(cp() , cp(*A*,*list A*)) .

The operations on lists may be defined as follows:

<u>def</u> *null,nonnull* = *predicates* (*list A*) ,
<u>def</u> *nullc,nonnullc* = *parts* (*list A*) ,
<u>def</u> *h,t* = *selectors nonnullc* ,
<u>def</u> *prefix x y* = *construct nonnullc*(*x,y*) ,
<u>def</u> *nullist* = *construct nullc* () ,

in which *head* has been abbreviated to *h*, and *tail* to *t*. The expression $x{:}y$ is used as an abbreviation for prefix x y, () denotes the null list, *a,b,c,d,e* is used as an alternative to a: (b: (c: (d: (e: ()))), and u x denotes a list with one element. The types of the list functions introduced are:

null ∊ (*A-list* → *truth value*) ,
head ∊ (*nonnull A-list* → *A*) ,
tail ∊ (*nonnull A-list* → *A-list*) ,
prefix ∊ (*A* → (*A-list* → *A-list*)) ,
() ∊ *A-list* .

These functions are closely interrelated as follows:

$null() = \text{true}$,

$null(x{:}y) = \text{false}$,

$h(x{:}y) = x$,

$t(x{:}y) = y$,

$(h\ z)\ :\ (t\ z) = z$,

in which x is an A, y is an *A-list*, and z is a *nonnull A-list*. Each structure description is assumed to create functions that conform to a number of axioms of this type.

Many functions that operate on lists have the same basic structure, as can be seen from the following:

<u>def</u> *sum x* =
  if *null x*
  then 0
  else (*h x*) + *sum* (*t x*)

e.g., *sum* (1,2,3,4) = 10 ,

<u>def</u> *product x* =
  if *null x*
  then 1
  else (*h x*) × *product* (*t x*)

e.g., *product* (1,2,3,4) = 24 ,

<u>def</u> *append x y* =
  if *null x*
  then *y*
  else (*h x*) : (*append* (*t x*)*y*)

e.g., *append* (1,2)(3,4,5) = 1,2,3,4,5 ,

<u>def</u> *concat x* =
  if *null x*
  then ()
  else *append* (*h x*)(*concat* (*t x*))

e.g., *concat*((1,2), (3,4), ()) = 1,2,3,4 ,

<u>def</u> *map f x* =
  if *null x*
  then ()
  else (*f*(*h x*)) : (*map f*(*t x*))

e.g., *map square* (1,2,3,4) = 1,4,9,16 .

It can be seen that these functions only differ in the first arm of the conditional expression and in the function that combines whatever is produced from the head of the list (usually the head itself) with whatever is produced by applying the same function to the tail of the list. The common part of these functions may be expressed as a function called *list1*, defined below, in which the parts that are not common have been made variables.

<u>def</u> *list1 a g f x* =
  if *null x*
  then *a*
  else *g*(*f*(*h x*)) (*list1 a g f* (*t x*))

The result of applying the function *list1 a g f* to a list is *a* if the list is empty; otherwise it is the result of applying *g* to two arguments: 1) the result of applying *f* to the head of the list, and 2) the result of applying *list1 a g f* to the tail of the list. It is now possible to redefine the five functions in terms of *list1*. It can be seen that this technique both saves writing and is more likely to produce a correct program because the complex programming (i.e., the conditional expression and looping) has been written once and for all in the function *list1*. The *list1* function is similar to the reduction operator in APL [6]; thus +/ in APL is similar to *list1 0 plus I*. The following definitions use *postfix*, which adds a new item to the end of a list, defined as follows:

<u>def</u> *postfix x y* = *append y* (*u x*) .

Also I is the identity function and $K\ x\ y = x$.

def $sum = list1\ 0\ plus\ I$,

def $product = list1\ 1\ mult\ I$,

def $append\ x\ y = list1\ y\ prefix\ I\ x$,

def $concat = list1\ ()\ append\ I$,

def $map\ f = list1\ ()\ prefix\ f$.

Some other examples are:

def $length = list1\ 0\ plus\ (K\ 1)$,

def $sumsquares = list1\ 0\ plus\ square$,

def $reverse = list1\ ()\ postfix\ I$,

def $identity = list1\ ()\ prefix\ I$.

The types of arguments of *list1* may be deduced from their pattern of application as follows. The whole function *list1 a g f* must operate on a list. Let us assume that it is an *A-list* and that it produces a member of B, i.e.

$(list1\ a\ g f) \in (A\text{-}list \rightarrow B)$.

Then the argument a must be a B. The function f operates on an A and let us assume it produces a C:

$f \in (A \rightarrow C)$.

Then g must operate on a C and the result of applying *list1 a g f* to an *A-list*, which is a B, must produce a B. The type of g is therefore

$g \in (C \rightarrow (B \rightarrow B))$.

The function *list1* has the property

$list1\ a\ g\ f\ (append\ x\ y) = list1\ (list1\ a\ g\ f\ y)\ g\ f\ x$

provided that the types of a, g, and f conform to the scheme above for some sets A, B, and C, and that x and y are both *A-lists*. This may be proved in a mechanical fashion from the axioms for lists and from the definitions of *list1* and *append*. The property emphasizes the fact that in all functions defined in terms of *list1*, the list is first scanned to its end, and then the actions g are taken on the way back. The same result may be obtained in two ways: 1) by concatenating two lists x and y, and then applying *list1 a g f* to the result, and 2) by first applying *list1 a g f* to the second list (y), and then using the result as an initial value (the a argument) for the same function when it is applied to the first list (x).

Two cases must be examined in order to prove the following property of *list1*:

$list1\ a\ g\ f\ (append\ x\ y) = list1\ (list1\ a\ g\ f\ y)\ g\ f\ x$.

One is when x is the null list and the other when x is non-null. The proof proceeds by reducing each side to the same expression. First substitute *list1 y prefix I x* for *append x y*, and then let:

$\text{LHS} = list1\ a\ g\ f\ (list1\ y\ prefix\ I\ x)$

$\text{RHS} = list1\ (list1\ a\ g\ f\ y)\ g\ f\ x$.

Then if x is the null list,

$\text{LHS} = list1\ a\ g\ f\ (list1\ y\ prefix\ I\ ())$

$\quad = list1\ a\ g\ f\ y$,

and

$\text{RHS} = list1\ (list1\ a\ g\ f\ y)\ g\ f\ ()$

$\quad = list1\ a\ g\ f\ y$.

Both steps are applications of $list1\ a\ g\ f\ () = a$. When x is not null, it takes the form $h\ x{:}t\ x$, so that

$$\begin{aligned}\text{LHS} &= list1\ a\ g\ f\ (list1\ y\ prefix\ I\ (h\ x{:}t\ x))\\ &= list1\ a\ g\ f\ (prefix\ (I(h\ x))\ (list1\ y\ prefix\ I\ (t\ x)))\\ &= list1\ a\ g\ f\ (prefix\ (h\ x)\ (list1\ y\ prefix\ I\ (t\ x)))\\ &= g\ (f(h\ x))\ (list1\ a\ g\ f(list1\ y\ prefix\ I\ (t\ x)))\\ &= g\ (f(h\ x))\ (list1\ (list1\ a\ g\ f\ y)\ g\ f\ (t\ x)),\end{aligned}$$

and

$$\begin{aligned}\text{RHS} &= list1\ (list1\ a\ g\ f\ y)\ g\ f\ (h\ x{:}t\ x)\\ &= g\ (f(h\ x))\ (list1\ (list1\ a\ g\ f\ y)\ g\ f\ (t\ x))\\ &= LHS.\end{aligned}$$

Each step is an application of

$list1\ a\ g\ f\ (h\ x{:}t\ x) = g\ (f\ (h\ x))\ (list1\ a\ g\ f\ (t\ x))$

except the last in the reduction of the LHS, in which the property is assumed to be true for the tail of the list. All the other properties of functions that follow can be proved in the same mechanical fashion.

The functions defined in terms of *list1* all scan the list (i.e., accumulate the result) from right to left. There is a second family of functions whose members scan lists from left to right, which can be defined by using a function called *list2*.

```
def list2 a g f x =
        if null x
        then a
        else list2 (g(f(h x))a) g f (t x)
```

The arguments a, g, f, and x must be the same types as those for *list1*. The function *list2* has a property analogous to that of *list1*, namely:

$list2\ a\ g\ f\ (append\ x\ y) = list2\ (list2\ a\ g\ f\ x)\ g\ f\ y$.

Thus we may either append two lists x and y and then apply *list2 a g f* to the result, or first apply the function to x and then use the result as an initial value for the same function when it is applied to y. Note that the roles of x and y are reversed in the properties of *list1* and *list2*. In

the case of functions defined using *list2*, the result is accumulated as the list is scanned from left to right.

The function *list2* can be implemented by the "iterative" program below; thus it may be more efficient to use *list2* than *list1*.

```
L: if null x
   then a
   else
        a: = g (f (h x)) a
        x: = t x
        go to L
```

It is also possible to prove that

list1 a g f x = *list2 a g f* (*reverse x*) .

This means that *list1 a g f* produces the same result when applied to a list x as *list2 a g f* does when it is applied to the reversal of list x. Since

append x y = *list1 y prefix x*

= *list2 x postfix y*

and

reverse = *list1* () *postfix I*

= *list2* () *prefix I* ,

it is possible by substituting particular values for a, g, and f in the relationship between *list1* and *list2* to prove

reverse(*reverse x*) = x ,

append(*append x y*) *z* = *append x*(*append y z*) ,

reverse(*append x y*) = *append*(*reverse y*)(*reverse x*) .

The generating function (g.f.) for an *A-list* may be obtained immediately from the structure description for lists

list A = du(cp.(),cp(*A*,*list A*)) ;

$list(x) = 1 + x\ list(x)$.

There is one list having no components (x^0 or 1) . A nonnull *A-list* has a head, which is an A, and a tail, which is an *A-list*. In the corresponding g.f., an x appears corresponding to the head A and a *list(x)* appears corresponding to *list A*, and these are multiplied together in the g.f. This may be solved, giving:

$$list(x) = 1/(1-x)$$
$$= 1 + x + x^2 + x^3 + x^4 + \cdots .$$

Another way to read the definition is: a list is either empty, or is a *1-list*, or is a *2-list*, etc., and the generating function for a list of length n is x^n.

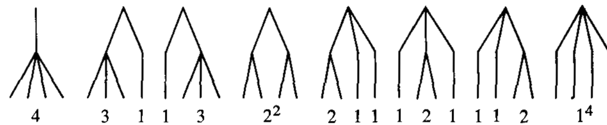


Figure 1 Compositions of 4.

The generating functions for lists whose lengths are restricted in some way can be written down immediately. For example, the nonnull lists have the generating function:

$$nlist(x) = x/(1-x) = x + x^2 + x^3 + x^4 + \cdots .$$

and the g.f. for all lists no longer than j is

$$(1 - x^{j+1})/(1-x) .$$

The method for creating new generating functions by substitution may be illustrated as follows. The g.f. for a nonempty list of nonempty lists is *nlist* (*nlist* (x)), where $nlist = x/(1-x)$; therefore it is $x/(1-2x)$, and there are 2^{n-1} configurations with n atomic components. The structures are usually called *compositions*, and the 8 compositions of 4 are given in Fig. 1.

The associated function for scanning a nonempty list is:

```
def nlist g f x =
        if null (t x)
        then f (h x)
        else g(f (h x)) (nlist g f (t x))
```

and the function for scanning compositions is *nlist* g_1 (*nlist* g_2 f_2). This can be extended to multipart compositions as follows. An (*A-list*)-(*B-list*) pair, i.e., having a first, which is an *A-list*, and a second, which is a *B-list*, has the generating function

$$(1/1-x)\ (1/1-y) ,$$

and if the pair ((), ()) is omitted, the generating function becomes

$$(1/1-x)\ (1/1-y) - 1$$

or

$$(x/1-x) + (y/1-y) + (x/1-x)\ (y/1-y) .$$

This corresponds to the structure

An *A-B npair* is
either an *A-nlist*
or a *B-nlist*
or an (*A-nlist*)-(*B-nlist*) *pair*.

The corresponding program could be written

```
def npair p f1 g1 f2 g2 (x,y) =
        if null x
        then nlist f1 g1 y
```

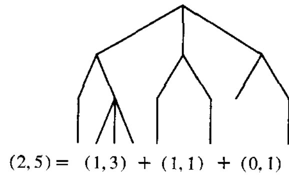


Figure 2 A two-part composition of (2.5).

else if *null y*
then $nlist\, f_2\, g_2\, x$
else $p(nlist\, f_2\, g_2\, x)\ (nlist\, f_1\, g_1\, y)$

A nonempty list of such pairs therefore has the generating function

$$nlist\ ((1/1-x)\ (1/1-y)-1)\ ,$$

in which the coefficient of $x^n y^m$ is the number of the two-part compositions of the two-part number (n,m). An example of a two-part composition of (2,5) is given in Fig. 2. The associated function for scanning two-part compositions is $nlist\ g_3\ (npair\ p\, f_1\, g_1 f_2\, g_2)$.

List structures

An *A-liststructure* is either atomic and is an *A* or is an (*A-liststructure*)*-list*. Thus

def *liststructure A* = du(*A*,*list* (*liststructure A*)) ,

def *atomic,nonatomic* = *predicates*(*liststructure A*) .

The functions *list1* and *list2* can be extended to list structures as follows:

```
def ls1 a g f x =
        if atomic x
        then f x
        else list1 a g (ls1 a g f) x
def ls2 a g f x =
        if atomic x
        then f x
        else list2 a g (ls2 a g f) x
```

In both, the same function *ls1 a g f* or *ls2 a g f* is applied to a component list structure as is applied to the list structure itself. In this case *f* and *ls1 a g f* must produce the same type of object, and $g \in B \rightarrow (B \rightarrow B)$. It follows that some of the functions defined for lists, i.e., those with $g \in B \rightarrow (B \rightarrow B)$ for some *B*, may be extended to be applicable to list structures. For example

def *sumls* = *ls1 0 plus I*

= *ls2 0 plus I* ,

def *productls* = *ls1 1 mult I*

= *ls2 1 mult I* ,

def *concatls* = *ls1* () *append u* ,

def *mapls f* = *ls1* () *prefix f* ,

def *lengthls* = *ls1 0 plus* (*K 1*) ,

def *sumsquaresls* = *ls1 0 plus square* ,

def *reversels* = *ls1* () *postfix I* ,

def *identityls* = *ls1* () *prefix I* .

The first two functions take the sum and product of the atomic components of a list structure, *concatls* flattens the list structure into a list, (*mapls f*) applies *f* to all the atomic components and produces a list structure with the same shape as the original containing the results, *lengthls* counts the number of atomic components, and *reversels* reverses all the lists in the structure. Examples of the applications of these functions follow:

let *ls* = (1,(2,(3,4)),5) ,

sumls ls = 15 ,

productls ls = 120 ,

concatls ls = (1,2,3,4,5) ,

mapls ls = (1,(4,(9,16)),25) ,

lengthls ls = 5 ,

sumsquares ls = 55 ,

reversels ls = (5,((4,3),2),1) ,

identityls ls = (1,(2,(3,4)),5) .

The relation between *list1* and *list2* is extended to

ls1 a g f x = *ls2 a g f* (*reversels x*) .

The generating function for a list structure is given by

$$ls(x) = x + list\ (ls\ x)\ .$$

The existence of *0-lists* and *1-lists* implies that there is an infinite number of list structures containing *n* atomic elements. If *0-lists* and *1-lists* are disallowed, the list g.f. becomes $x^2/(1-x)$, and the g.f. for list structures without *0-* and *1-lists* is:

$$nls(x) = x + nls^2(x)/(1-nls(x))\ ,$$

which may be solved to give

$$2nls^2(x) - (1+x)nls(x) + x = 0$$

$$nls(x) = 1/4(1+x-\sqrt{1-6x+x^2})$$

$$= x + x^2 + 3x^3 + 11x^4 + 45x^5 + \cdots .$$

This g.f. enumerates the ways of bracketing a sum of *n* numbers, in which each bracket must include at least two expressions, but the entire expression is unbracketed. The first four instances are:

a ,

$a + b$,

$a + b + c$, $(a + b) + c$, $a + (b + c)$,

$a + b + c + d$, $(a + b + c) + d$, $a + (b + c + d)$,
$(a + b) + c + d$, $a + (b + c) + d$, $a + b + (c + d)$,
$(a + b) + (c + d)$, $((a + b) + c) + d$,
$(a + (b + c)) + d$, $a + ((b + c) + d)$,
$a + (b + (c + d))$.

Trees

List structures can be considered to be ordered, rooted trees with components only at their leaves, or end points. Another structure that is often useful is a tree that has components at every node. If interior and end points are to be treated in the same way, the most useful tree structure is defined as follows:

An *A-tree* has a root, which is an *A* ,
and a listing, which is an *A-forest* .

An *A-forest* is an *(A-tree)-list* .

The function used to construct a tree from an *A* and an *A-forest* is called *ctree*. The following operations are defined for trees and forests:

def *tree A* = cp(*A*,*forest A*) ,

def *forest A* = *list*(*tree A*) ,

def *root, listing* = *selectors*(*tree A*)

def *ctree* = *construct*(*tree A*) .

Two families of functions can be defined using *list1* and *list2*:

def *tree1 a g f x* = *f* (*root x*) (*forest1 a g f* (*listing x*)) ,

def *forest1 a g f x* = *list1 a g* (*tree1 a g f*) *x* ,

def *tree2 a g f x* = *f* (*root x*) (*forest2 a g f* (*listing x*)) ,

def *forest2 a g f* = *list2 a g* (*tree2 a g f*) .

In this representation, an end point is a tree whose listing is the null list. As in the case of list structures, the same function is applied to both a component tree and the whole tree, and the same function is applied to both a component forest and the whole forest.

The expression below is used to describe the tree in Fig. 3. The comma is used for lists and the semicolon is used for *ctree*.

A;(*B*;(*E*;(),*F*;(*G*;(),*H*;())),*C*;(),*D*;

(*I*;(*K*(),*L*;()*J*;())))

There are several ways of scanning the nodes of a tree. They can be expressed as functions for listing all the atomic components. The result of applying (*tree1* () *append prefix*) to the tree in Fig. 3 is

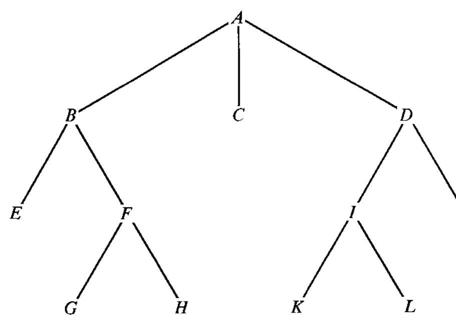


Figure 3 An example of a tree.

A B E F G H C D I K L J ,

and (*tree2* () *append prefix*) produces

A D J I L K C B F H G E ,

which is also the result of applying (*tree1* () *append prefix*) to the reversed tree.

A tree may be reversed, meaning that all its lists are reversed, by using:

def *reversetree* = *tree2* () *prefix ctree* ,

and a forest may be reversed by

def *reverseforest* = *forest2* () *prefix ctree* .

It is easy to prove that

reversetree(*reversetree x*) = *x*

and

reverseforest(*reverseforest x*) = *x*

Each function on a tree, defined in terms of *tree1* or *tree2*, may be applied to a forest by using the same arguments and the functions *forest1* and *forest2*, respectively, and vice versa. The property above may be generalized to give the following relationships between *tree1* and *tree2* and between *forest1* and *forest2* :

tree1 a g f x = *tree2 a g f* (*reversetree x*) ,

forest1 a g f x = *forest2 a g f* (*reverseforest x*) .

To determine the generating functions for trees and forests consider that:

$tree(x) = x\, forest(x)$,

$forest\ x = list(tree\ x)$.

Therefore

$$tree(x) = x\ list(tree(x))$$
$$= x/(1 - tree(x))$$

$tree^2(x) - tree(x) + x = 0$

$$tree(x) = \tfrac{1}{2}(1 - \sqrt{1-4x})$$
$$= \sum_{n\geq 1} 1/n \binom{2n-2}{n-1} x^n .$$
$$= x + x^2 + 2x^3 + 5x^4 + 14x^5 + 42x^6 + 132x^7 + \cdots .$$

Also since $tree(x) = x\, forest(x)$, then

$x\, forest^2(x) - forest(x) + 1 = 0$.

Forests with n nodes are in correspondence with the trees with $n + 1$ nodes formed by adding a root.

Binary trees

Two more restricted types of trees are considered next. Correspondences can be set up between both types of trees and the trees just defined. The first type of tree is called a *binary* tree, and can have 0 or 2 subtrees.

An *A-binary tree*
  is either empty
    or has a root, which is an *A* ,
      and a left and a right, both *A-binary trees.*

The following definitions apply to binary trees:

<u>def</u> *btree A* = du(cp(),cp(*A*,*btree A*,*btree A*)) ,

<u>def</u> *empty,nonempty* = *predicates* (*btree A*) ,

<u>def</u> *emptyc,nonemptyc* = *parts* (*btree A*) ,

<u>def</u> *root,left,right* = *selectors nonemptyc* ,

<u>def</u> *cbtree* = *construct nonemptyc* ,

<u>def</u> *etree* = *construct emptyc* () .

The function for constructing nonempty trees is called *cbtree*, and *etree* is the name of the empty binary tree.

The family of functions that scans binary trees can be produced by using:

```
def btree1 a g f x =
        if empty x
        then a
        else g (f (root x))
               (btree1 a g f (left x))
               (btree1 a g f (right x))
```

in which if $(btree1\ \ a\ g f) \in A\text{-}binary\ \ tree \to B$, then $a \in B$, $f \in A \to C$, and $g \in C \to (B \to (B \to B))$. It is possible to define another function on binary trees, which interchanges the roles of the left and right subtrees, as follows:

```
def btree2 a g f x =
        if empty x
        then a
        else g (f (root x))
               (btree2 a g f (right x))
               (btree2 a g f (left x))
```

The function for reversing a binary tree is therefore:

```
def reversebtree = (btree2 etree cbtree I)
                 = (btree1 etree (C ctree) I)
                   where C f x y = f y x .
```

There is a correspondence between a binary tree and a forest, which can perhaps best be seen by unwinding, so to speak, the structure definition of a forest.

An *A-forest* is an (*A-tree*)*-list*
  i.e., is either *null*
    or has a *h*, which is an *A-tree*,
      and a *t*, which is an (*A-tree*)*-list* .

In other words,
An *A-forest* is either *null*
    or has a (*root.h*), which is an *A*,
      and a (*listing.h*), which is an *A-forest*,
      and a *t*, which is an *A-forest* .

This definition has the same structure as a binary tree, and the correspondence can be expressed as the following correspondence between the functions on binary trees and forests:

binary tree	forest
empty	*null*
etree	()
root	*root.h*
left	*listing.h*
right	*t*
btree x y z	(*ctree x y*):*z*

Thus every function that either operates on or constructs binary trees may be transformed into a function that operates on or constructs forests. An example of this correspondence between forests and binary trees is shown in Fig. 4.

There is a transformation of forests that corresponds to a reversal of the corresponding binary tree. It is clear that the *forest1* function can be "unwound" in the same way that the structure definition was to give:

```
forest1 a g f x = list1 a g (tree a g f)x
                = if null x
                  then a
                  else g(tree1 a g f(h x))
                        (list1 a g(tree1 a g f(t x)))
```

$= \text{if } \mathit{null}\ x$
$\quad \text{then } a$
$\quad \text{else } g(f(\mathit{root}(h\ x))$
$\qquad\qquad (\mathit{forest1}\ \ a\ g\ f(\mathit{listing}(h\ x))))$
$\qquad\quad (\mathit{forest1}\ \ a\ g\ f(t\ x))$

Another scanning function for forests called *forest3* can be defined by interchanging the roles of the two forests at positions *listing.h* and *t* in this definition, giving:

$\mathit{forest3}\ \ a\ g\ f\ x =$
$\quad \text{if } \mathit{null}\ x$
$\quad \text{then } a$
$\quad \text{else } g(f(\mathit{root}(h\ x))$
$\qquad\qquad (\mathit{forest3}\ \ a\ g\ f(t\ x)))$
$\qquad\quad (\mathit{forest3}\ \ a\ g\ f(\mathit{listing}(h\ x)))$

The function for "rotating" a forest, i.e., producing the forest corresponding to the reversal of the binary tree underlying the original, is

<u>def</u> *rotate = forest3 () ctree prefix*.

It follows that

$\mathit{forest1}\ \ a\ g\ f\ x = \mathit{forest3}\ \ a\ g\ f(\mathit{rotate}\ x),$

$\mathit{rotate}(\mathit{rotate}\ x) = x.$

If *forest2* is given the same treatment as *forest1*, then the function *forest4* is produced having the properties:

$\mathit{forest2}\ \ a\ g\ f\ x = \mathit{forest4}\ \ a\ g\ f(\mathit{rotate}\ x),$

$\mathit{forest1}\ \ a\ g\ f\ x = \mathit{forest4}\ \ a\ g\ f(\mathit{rotate}(\mathit{reverse}\ x)).$

The rotate operation interchanges the two forests in Fig. 5. For a further discussion of this rotate operator see Holt [7].

A forest may be transformed into a binary tree by the function:

forest1 etree g pair where $g(x,y)z = \mathit{cbtree}\ x\ y\ z$

and $\mathit{pair}\ x\ y = x,y$.

A binary tree can be transformed into a forest by

btree1 () *g I* where $g\ x\ y\ z = (\mathit{ctree}\ x\ y):z$.

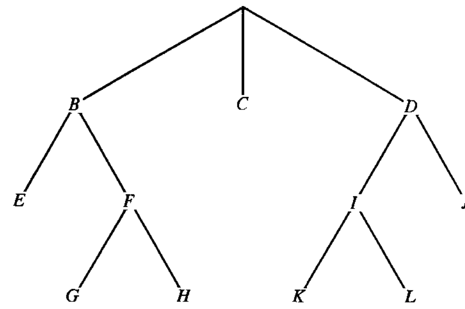
The generating function for binary trees is

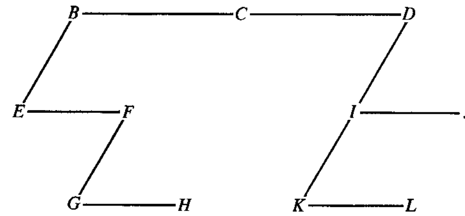
$\mathit{btree}(x) = 1 + x\ \mathit{btree}^2(x)$,

and therefore is equal to $\mathit{forest}(x)$.

Combinations

Another correspondence can be set up between trees and a structure called a *combination*. Combinations are trees with components only at their end points, and each tree can have either 0 or 2 subtrees.





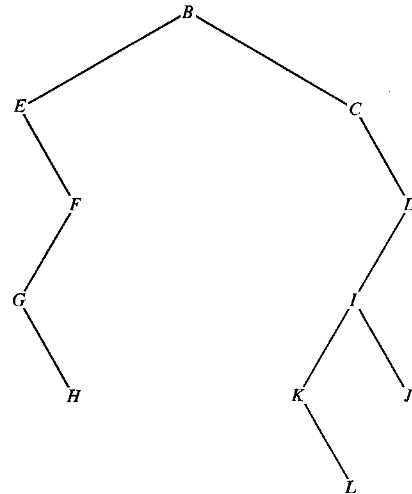
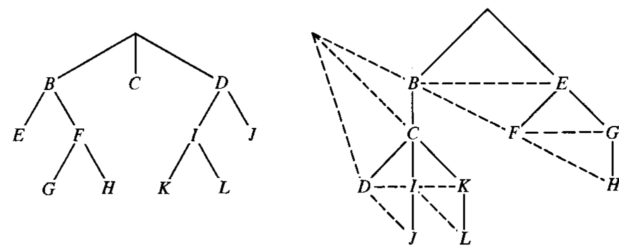


Figure 4 A correspondence between a forest and a binary tree.

Figure 5 Rotating a forest.



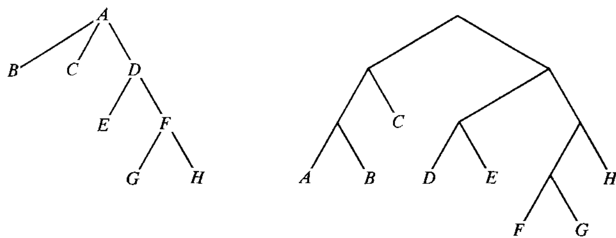


Figure 6 A correspondence between trees and combinations.

An *A-combination* is
either atomic and is an *A*
or has a left and a right, both *A-combinations*.

The following operations are defined for combinations:

def *comb A* = du(*A*,cp(*comb A*,*comb A*)),

def *atomic,nonatomic* = *predicates*(*comb A*),

def *atomicc,nonatomicc* = *parts*(*comb A*),

def *left,right* = *selectors nonatomicc*,

def *combine* = *construct nonatomicc*,

```
def combl g f x =
        if atomic x
        then f x
        else g(combl g f(left x))
              (combl g f(right x))
```

and *comb2* is defined similarly by interchanging *left* and *right*.

The interior nodes of a combination with n nodes form a binary tree with $n-1$ nodes. Both combinations and binary trees can be considered to be special cases of a structure in which the interior and end points are treated as two different types of atomic component.

An *A-B tree structure* is
either atomic, and is an *A*,
or has a root, which is a *B*,
and a left and a right,
both *A-B tree structures*.

The binary tree is formed by setting *A* empty, and the combination is formed by setting *B* empty.

There is also a correspondence between trees and combinations, and a function *tree5* can be written that scans the tree as if it were the corresponding combination. In this correspondence, a tree with a null listing corresponds to an atomic combination; when nonnull the tree at the head of the listing corresponds to the left of the combination, and the tree formed from the root and the tail of the listing corresponds to the right of the combination.

```
def tree5 g f x =
        if null(listing x)
        then f(root x)
        else g(tree5 g f(h(listing x)))
              (tree5 g f(ctree(root x)(t(listing x)))) .
```

A tree can be transformed to a combination by using (*tree5 combine I*) and a combination transformed to a tree by using

combl g u

where $g\ x\ y = ctree(root\ y)(x{:}listing\ y)$

and $u\ x = x{:}()$.

An example of this correspondence is given in Fig. 6, in which both trees have been reversed. The correspondence may be interpreted as a translation from a functional notation in which arguments are listed, i.e., *A*(*B*,*C*,*D*(*E*,*F*(*G*,*H*))) to one in which only application of a function to a single argument is used, i.e., ((*A B*)*C*) ((*D E*)((*F G*)*H*)).

The generating function for combinations is

$$combination(x) = x + combination^2(x)$$

and so *combination*(*x*) has the same recurrence relation as *tree*(*x*).

Weights of trees

The weight of a tree with respect to a certain type of atomic component is the sum of the lengths of the paths from each atomic component to the root. The generating functions for the weights of trees can also be obtained almost automatically from the structure definition.

Suppose that $T(x,y)$ is a generating function for a set of trees in which the coefficient of $x^n y^w$ is the number of trees with n nodes and weight w. If this set of trees is moved down one level, then one must be added to the weight for each atomic component in the tree, or for each x in the generating function. The generating function for the set of trees moved down one level is therefore $T(xy,y)$, assuming that x is the variable that corresponds to the atomic components whose depths are to be counted in the weight. Therefore the generating functions for the weights of trees can be written down directly from their structure descriptions.

The weight generating functions for trees, binary trees, and combinations are therefore:

$$wtree(x,y) = x/(1 - wtree(xy,y)),$$

$$wbtree(x,y) = 1 + x\ wbtree^2(xy,y),$$

$$wcombination(x,y) = x + wcombination^2(xy,y)\ .$$

To obtain the expected weight of a set of trees with n nodes, first $T(x,y)$ is differentiated with respect to y and then y is set equal to one. This has the effect of multiplying the number of trees with weight w and n nodes by the weight w.

If

$$T(x,y) = \sum_{n,w} t_{nw} x^n y^w,$$

then

$$T_y(x,y) = \sum_{n,w} w t_{nw} x^n y^{w-1},$$

and

$$T_y(x,1) = \sum_n \left(\sum_w w t_{nw} \right) x^n .$$

The number of trees with n atomic components is the coefficient of x in $T(x,1)$. If $T(x,1) = \sum t_n x^n$, then the expected weight of trees with n nodes is $\sum w t_{nw} / t_n$. If *wcombination* is abbreviated to c, then the recurrence relation for combinations can be differentiated with respect to y to give

$$2c(xy,y)(c_y(xy,y) + xc_x(xy,y)) = c_y(x,y) .$$

Therefore $c_y(x,1) = 2xc(x,1)c_x(x,1)/1 - 2c(x,1)$

$$= x(1 - \sqrt{1-4x})/(1-4x) .$$

Hence the sum of the weights of combinations with $n+1$ nodes is $4^n - \binom{2n}{n}$. By a similar argument, the sum of the weights of all the trees with $n+1$ nodes or forests with n nodes is $(1/2)(4^n - \binom{2n}{n})$, i.e., just half of that for combinations. The sum of the weights of binary trees with n nodes is $4^n - (3n+1)/(n+1)\binom{2n}{n}$.

Summary

The four-way correspondence among 1) descriptions of sets, 2) general-purpose functions for scanning a member, 3) the generating functions for counting the number of members with n atomic components, and 4) the generating functions for the weights of trees is applicable to any set that can be described in terms of du, cp, and existing sets. The most general case is a number of mutually recursive definitions of sets, which give rise to a number of mutually recursive functions and a number of mutually recursive generating functions, which all have the same pattern of definition and describe trees having certain restrictions.

Although it is a straightforward matter to obtain a generating function from the structure description, it is often difficult to obtain an explicit expression for the coefficients. One general method that is applicable to this situation is the method of Lagrange's expansion, as extended to the multivariable case by Good [8].

It seems likely that there are other close correspondences between programming and combinatorial theory in which the mathematical operations used to construct generating functions correspond to the methods of piecing together parts of programs, and the methods for obtaining the coefficients of variables of generating functions correspond to the actions of the programs in constructing the different configurations that are enumerated by the generating function. One example is the relationship between labeled trees and "reluctant functions," as described by Mullin and Rota [9]. Another is the relationship between the details of the performance of differential operators that select coefficients of generating functions and programs that construct the related configurations, as described by MacMahon [10].

This paper contains some of the material from a chapter of a forthcoming book entitled "Advanced Systems Programming Techniques" to be published by the Addison-Wesley Publishing Co. as one volume in the *IBM Systems Programming Series* of books.

References

1. J. McCarthy, "A Basis for a Mathematical Theory of Computation," *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg, North-Holland Publishing Co., Amsterdam (1963) 33–70.
2. P. J. Landin, "A Correspondence between Algol 60 and Church's Lambda-Notation," Comm. ACM **8,** No. 2, 89–101, 158–159 (1965).
3. P. Lucas and K. Walk, "On the Formal Description of PL/1," *Annual Review in Automatic Programming* **6,** Part 3, 105–181 Pergammon Press, New York (1969).
4. A. van Wijngaarden, "Report on the Algorithmic Language Algol 68, *Numerische Mathematik* **14,** 79–218 (1969).
5. H. B. Curry and R. Feys, *Combinatory Logic*, North-Holland Publishing Co., Amsterdam (1958).
6. K. E. Iverson, A *Programming Language*, John Wiley and Sons, New York (1962).
7. A. W. Holt, "A Mathematical and Applied Investigation of Tree Structures," Ph.D. Thesis, University of Pennsylvania (1963).
8. I. J. Good, "The Generalization of Lagrange's Expansion and the Enumeration of Trees," *Proc.Camb.Phil.Soc.* **61** (1965) 499–517.
9. R. Mullin and G-C. Rota, "On the Foundations of Combinatorial Theory III of Binomial Enumeration," in *Graph Theory and its Applications*, edited by B. Harris, Academic Press (1970) 167–213.
10. P. A. MacMahon, *Combinatory Analysis*, Cambridge University Press, Vol 1 (1915), vol 2 (1916).

Received April 12, 1972

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.