Recursive Definitions

Almost all mathematical structures are defined by induction (recursion). The *composition* of valid compound objects and the *decomposition* of compound objects into elementary components is a recursive process. The definitions of *domain functions* are recursive. Most *proof systems* use induction for verification. Recursion is also necessary to write *elegant programs*.

An inductive definition consists of three components:

- a ***base case***, the simplest possible application of the induction
- an ***inductive case*** which assumes an arbitrary member of the domain, and constructs the adjacent member.
- an ***ordering principle*** which provides a structure for inferring that when one member can be constructed from adjacent member, then all members can be constructed.

Learning to Think Recursively

Recursive programming requires a different way of thinking about and writing computer programs. Procedural thinking was necessary twenty years ago when the art of programming was much closer to the construction of the OS and its hardware infrastructure. In this century, the programmer must *address the problem rather than the mechanism*. John McCarthy, the father of LISP and symbolic programming, says it nicely:

> "In order to write recursive function definitions, one must think about programming differently than is customary when writing programs in languages like FORTRAN or ALGOL [or C] or in machine language. In these languages, one has in mind the state of the computation as represented by the values of certain variables or locations in the memory of the machine, and then one writes statements or machine instructions in order to make the state change in an appropriate way. When writing recursive function definitions one takes a different approach. Namely, one thinks about the value of the function, asks for what values of the arguments the value of the function is immediate [the base case], and, given arbitrary values of the arguments, for what simpler arguments must the function be known in order to give the value of the function for the given arguments [the inductive case]."
>
> -- John McCarthy, LISP Programming and Proving

Induction Principles

When the three components of an inductive definition are combined, they produce an ***Induction Principle*** for the particular domain. Inductive principles are *second order functions, or functionals*. We are most familiar with first-order functions, which vary over domain objects (ie variables). Second-order functions vary over other functions; the domain is a set of functions rather than a set of objects. Sometimes second-order functions are called *functional*

schema, they are patterns, or schema, which specify relations between objects which hold for a *set* of functions. Each schema can be instantiated to many (often infinite) specific induction rules for specific functions.

Different object domains have different induction principles. However, across most domains, the form of the inductive principles is quite similar. Thus we can even consider abstraction of inductive schema over domains. For example,

Domain	*Decomposition*	*Base*	*Inductive Step*
integers	`-1`	`0`	`f[n+1] -> f[n]`
sets	`choose, rest`	`{ }`	`f[choose·rest] -> f[rest]`
logic	`antecedent, consequent`	`F`	`(p->q) -> q`
pairs	`first,second`	`<nil,nil>`	`f[first] -> f[second]`
lists	`first, rest`	`( )`	`f[first·rest] -> f[rest]`
strings	`prefix, rest`	`Empty char`	`f[prefix·rest] -> f[rest]`
binary trees	`left,right`	`root`	`f[left·right]->f[left]&f[right]`
trees	`left, butleft`	`root`	`f[left·butleft]->f[left]&f[butleft]`

Generic Induction Schema

Primitive recursive schema without parameters for the ***integer domain***:

```
f[n] =def=
            if n=0 then k else h[n-1,f[n-1]]
```

A function acting on an integer argument `n` can be generically thought of as follows:

If `n` is `0`, return some constant `k`,
otherwise apply function `h` with the arguments one step closer to `0`
[that is, with arguments `(n-1)` and `f` applied to `(n-1)`].

Example: the factorial function

```
f = fac     k = 1       h = *       h[p,q] = (p+1)*q

fac[n] =def=
            if n=0 then 1 else n*fac[n-1]
```

Primitive recursive schema with parameters for the ***integer domain***
(for simplicity, only one parameter is shown here):

```
f[n,m] =def=
            if n=k then g[m] else h[n-1,m,f[n-1,m]]
```

Example: the times function

```
f = *       k = 1       g[m] = m    h = +       h[p,q,r] = q*(r+1)

*[n,m] =def=
            if n=1 then m else +[m,*[n-1,m]]
```

Primitive recursive schema without parameters for the ***list domain***:

```
f[x] =def=
            if null[x] then k else h[first[x],rest[x],f[rest[x]]]
```

If the list is empty, return constant `k`,
otherwise apply `h` with arguments one step closer to the empty list.

Example: the last function

```
f = last    k = first[x]      h = identity      h[p,q,r] = r

last[x] =def=
            if null[rest[x]] then first[x] else last[rest[x]]
```

Example: the length function

```
f = len     k = 0             h = +1            h[p,q,r] = r

len[x] =def=
            if null[x] then 0 else +1[len[rest[x]]]
```

Example: the reverse function (`*` is the append function)

```
f = reverse       k =()       h = append        h[p,q,r] = r*(p)

reverse[x] =def=
   if null[x] then () else (reverse[rest[x]] * list[first[x]])
```

Primitive recursive schema with parameters for the ***list domain***:
Here, the parameter `y` used in the definition, but not decomposed by the induction.

```
f[x,y] =def=
        if null[x] then g[y] else h[first[x],rest[x],y,f[rest[x],y]]
```

If the list is empty, return `g` applied to parameter `y`
otherwise apply `h` to the decomposed list, to the parameter,
and to `f` applied to the rest of the list.

Example: the member function (`u` is the parameter)

```
f = member  g = False   h = or            h[p,q,r,s] = r=p or s

member[u,x] =def=
            if empty[x] then False
                  else (u=first[x] or member[u,rest[x]])
```

Primitive recursive schema without parameters for the ***tree domain***:

```
f[x] =def=
      if leaf[x] then g[x] else h[first[x],rest[x],f[first[x]],f[rest[x]]]
```

If the tree is an leaf, return `g[x]`
else apply `h` to the decomposed tree and to `f` applied to the decomposed tree.

Example: the size function (number of nodes in the tree, including leaves)

```
f = size    g[x] = 1    h = +       h[p,q,r,s] = r + s

size[x] =def=
            if leaf[x] then 1
                  else (size[first[x]] + size[rest[x]])
```

Primitive recursive schema with parameters for the ***tree domain***:

```
f[x,y] =def=
        if leaf[x] then g[x,y] else
              h[first[x],rest[x],y,f[first[x],f[first[x],first[y]]],
                                   f[rest[x],f[rest[x],rest[y]]]]
```

If the tree is a leaf, return `g[x,y]`
otherwise apply `h` with arguments of the decomposed tree,
(`f` applied to the first of the tree and to `f` applied to the first of each argument `x,y`),
and (`f` applied to the rest of the tree and to `f` applied to the rest of each argument `x,y`)

Example: the tree-equal function

```
f = tree-equal    g = =       h[p,q,r,s,t] =

tree-equal[x,y] =def=
            x=y or (not[leaf[x]] and not[leaf[y]]
                        and tree-equal[first[x],first[y]]
                        and tree-equal[rest[x],rest[y]])
```

Recursive Mathematical Structures

Below isa partial listing of the recursive definitions of some common mathematical structures and their functions. In the following, I have used the token "•" to stand for a generic constructor.

Propositional Logic

Notation:
truth symbols = {T,F}
propositional symbols = {a,b,...}
sentence symbols = {A,B,...}

Sentences:
Truth symbols are sentences.
Propositional symbols are sentences.
If A and B are sentences, so is A->B.

Truth-values:
The truth-values of T and F are respectively True and False.
The truth-value of a is the value assigned to it by the semantics.
The truth-value of A->B is True when either A is False
or when both A and B are True.

Predicate Logic

Notation:
truth symbols = {T,F}
constant symbols = {a,b,...}
variable symbols = {x,y,...}
function sysmbols = {f,g,...}
relation symbols = {p,q,...}

Terms:
Constants and variables are terms.
If {t1,t2,...} are terms, and f has arity n, then f[t1,...,tn] is a term.

Relations:
Truth symbols are relations.
If {t1,t2,...} are terms, and p has arity n, then p[t1,...,tn] is a relation.

Sentences:
Terms are sentences.
Relations are sentences.
If S1 and S2 are sentences, then so is S1->S2.
If S is a sentence, then so is (all x.S)

Well-formed Parentheses (WFP)

() is a WFP
If x and y are WFP, so is (x) and xy

Pairs

If x and y are both atoms, then <x,y> is a pair.

Strings

E is a string. (E is the empty character)
u is a string. (u is a character)
If x is a string, then so is u·x

Binary Strings

E, 0, 1 are each binary strings.
If x is a binary string, so is x·0 and x·1.

Lists

() is a list.
If x is a list, so is u·x (u is an atom)

```
last[()] = ()
last[(u)] = u
last[u·x] = last[x]

length[()] = 0
length [u·x] = length[x] + 1

member[u,()] = F
member[u,v·x] = u=v or member[u,x]

copy[()] = ()
copy[u·x] = u·copy[x]

insert[u,()] = (u)
insert[u,v·x] = if u =< v then u·(v·x) else v·insert[u,x]

alternating-elements[()] = ()
alternating-elements[(u)] = (u)
alternating-elements[u·v·x] = u·alternating-elements[x]

reverse[()] = ()
reverse[u·x] = reverse[x]*(u)       (* is append)

same-length[(),()] = T
same-length[(),v·y] = F
same-length[u·x,()] = F
same-length[u·x,v·y] = same-length[x,y]
```

```
lintersection[(),()] = ()
lintersection[(),v·y] = ()
lintersection[u·x,()] = ()
lintersection[u·x,v·y] =
      if u=v then u·lintersection[x,y] else
            (lintersection[x,v·y] or lintersection[u·x,y]

lequal[(),()] = T
lequal[u·x,()] = F
lequal[(),v·y] = F
lequal[u·x,v·y] = u=v and lequal[x,y]

append[(),y] = y
append[v·x,y] = v·[append[x,y]

substitute[u,v,()] = ()
substitute[u,v,w·x] = if v=w then u·[substitute[u,v,x]]
      else [substitute[u,v,x]
```

Integers

Notation:

```
Successor[n] = n'
```

```
integer[1]
if integer[x] and integer[y] then integer[x+y]
```

Counting:

```
not[x'=0]
1      = 0'
n + 1  = n'
```

Addition:

```
m + 0  = m
m + n' = (m + n)'
```

Multiplication:

```
m * 0  = 0
m * n' = (m * n) + n
```

Exponentiation:

```
m ^ 0  = 1
m ^ n' = (m ^ n) * n
```

```
sum[0] = 0
sum[i'] = sum[i] + i'

fac[0] = 1
fac[i'] = fac[i] * i'

sumfac[0] = 1
sumfac[i'] = fac[i'] + sumfac[i]

fib[1] = fib[2] = 1
fib[i''] = fib[i'] + fib[i]
```

```
power-of-2[0] = 1
power-of-2[n'] = 2*power-of-2[n]

0 = 0
m' = n' iff m = n
```

Trees

Leaves are trees.
If `x` and `y` are trees, then so is `x·y`

```
depth[()] = 0
depth[u·x] = depth[x]
depth[(x)] = 1 + depth[x]

flatten[()] = ()
flatten[u] = (u)
flatten[x·y] = flatten[x]*flatten[y]

fringe[()] = ()
fringe[u] = u
fringe[u·y] = u·fringe[x]
fringe[x·y] = fringe[x]*fringe[y]

tsubstitute[x,y,()] = ()
tsubstitute[x,y,u] = if u=y then x else u
tsubstitute[x,y,z] =
tsubstitute[x,y,first[z]]*tsubstitute[x,y,rest[z]]

treverse[()] = ()
treverse[u·y] = treverse[y]*(u)
treverse[x·y] = treverse[y]*treverse[(x)]
```

Sets

`{}` is a set.
If `S` is a set, then so is `u·S`

```
powerset[{}] = {{}}
powerset[S+{e}] = powerset[S]*(S+e)

member[u,{}] = F
member[u,v·S] iff u=v or member[u,S]

add-to-set[u,{}] = {u}
add-to-set[u,v·S] = if u=v then S else add-to-set[u,S]

{} = {}
u·S1 = v·S2 iff u=v and S1=S2
```

```
union[{},S] = S
union[u·S1,S2] = if member[u,S2] then union[S1,S2]
                        else u·union[S1,S2]

intersection[{},S] = {}
intersection[S,{}] = {}
intersection[u·S1,S2] = if member[u,S2] then u·intersection[S1,S2]
                              else intersection[S1,S2]

subset[{},S2] = T
subset[S1,{}] = F
subset[u·S1,S2] = member[u,S2] and subset[S1,S2]

proper-subset[S1,S2] iff subset[S1,S2] and not[S1=S2]

cardinality[{}] = 0
cardinality[u·S] = if [member[u,S]] then cardinality[S] + 1
                              else cardinality[S]
```

Functions

```
andlist[()] = ()
andlist[u·x] = u and andlist[x]

map[f,()] = f[()]
map[f,u·x] = f[u]·map[f,x]

mapfilter[p,()] = ()
mapfilter[p,u·x] = if p[u] then u·mapfilter[p,x] else mapfilter[p,x]

every[p,()] = F
every[p,u·x] = p[u] and every[p,x]

apply[f,()] = f[()]
apply[f,u·x] =

count-if[p,()] = 0
count-if[p,u] = if p[u] then 1 else 0
count-if[p,u·x] = if p[u] then count-if[p,x] + 1 else count-if[p,x]

delete-if[p,()] = ()
delete-if[p,u·x] = if p[u] then delete-if[p,x] else u·delete-if[p,x]

ordered-by[p,()] = T
ordered-by[p,(u) = T
ordered-by[p,(u v)] = p[u,v]
ordered-by[p,u·v·x] = p[u,v] and ordered-by[p,v·x]

map-if[f,p,()] = if p[] then f[] else ()
map-if[f,p,u·x] = if p[u] then f[u]·map-if[f,p,x] else map-if[f,p,x]
```

Primitive Recursive Functions (PRF)

The successor function is a PRF. `s[i] = next[i]`
The zero function is a PRF. `z[i] = k`
The selector functions are PRFs. `first[i], rest[i]`
If `f` and `g` are PRF, then so is `(f o g)`. `function composition`
If `f` and `h` are PRF, then so is any function which fits a primitive recursion schema,
EG: `(if n=0 then z[n] else h[n-1,f[n-1]])`

Miscellaneous Applications

marking a ruler:

```
rule[left,right,0] = no-op
rule[left,right,height] = let m= (left+right)/2
                          mark[m,height];
                          rule[left,m,height-left]
                          rule[m,right,height-left
```

binary tree traversal

```
traverse[()] = no-op
traverse[leaf[node]] = no-op
traverse[node] = traverse[left[node]];
                 visit[node];
                 traverse[right[node]]
```

depth-first binary tree search

```
depthsearch[u,()] = F
depthsearch[u,v] = u=v
depthsearch[u,v·y] = if u=v then T else depthsearch[u,y]
depthsearch[u,x·y] = depthsearch[u,x] or depthsearch[u,y]
```

insertion sort

```
insertion-sort[()] = ()
insertion-sort[u·x] = insert[u, sort[x]]
```

symbolic differentiation (all in reference to `dx`)

```
dc = 0
dx = 1
d(-U) = -dU
d(U+V) = dU + dV
d(c*U) = c* dU
d(U*V) = u*dV + V*dU
d(U/V) = d(U*(V^-1))
d[U^c] = c*(U^(c-1))*dU
d(ln U) = (U^-1)*dU
```