Induction and Recursion

Induction is a mathematical proof technique. When this technique is used in programming, it is called **recursion**. Induction/recursion is the fundamental mechanism for

- extending logical proof techniques into object domains and data structures,
- defining and building mathematical and programming objects,
- decomposing objects so that functions can be applied to the elementary units, and
- robust programming style and program verification.

Many practical computational problems are most succinctly expressed in a recursive form (for instance, tree and graph traversal, spatial decomposition, divide-and-conquer algorithms, sorting, searching, and large classes of mathematical functions). As well, recursive function theory defines what can and cannot be computed.

Inductive Definition

An *inductive definition* consists of

1) a base case
2) a general generating case
3) an ordering principle which moves from one general case to the next

Base case: the value of **the most elementary case**

Examples:

`zero`	the additive identity
`one`	the multiplicative identity
`Phi`	the empty set
`nil`	the empty list, the empty tree
`false`	the logical ground

Generating rule: the transform which **defines the next case**, given an arbitrary case.

Examples:

```
power-of-2[n]     =     2*power-of-2[n-1]
summation[n]      =     summation[n-1] + n
prefix[str]       =     first-char · rest[str]
last[list]        =     rest[list] = nil
length[list]      =     length[rest[list]] + 1
member[x,S]       =     x=select[S] or member[x,rest[S]]
power-set[S]      =     power-set[S-{ele}]*S
node[btree]       =     left[btree] + right[btree]
logic-form[lf]    =     ante[lf] implies conseq[lf]
parenthesis[pf]   =     "("in[pf]")" or left[pf] + right[pf]
```

Recursive Programming

Inductive definitions build up from the base case to any degree of complexity. Recursive programs reduce any degree of complexity one step at a time until the base case is reached. A recursion must be **well-founded**, that is the steps must eventually terminate at the base. In most cases, the step increment is monotonically decreasing.

Recursive programs can be expressed in two forms, mathematical and accumulating. The mathematical form accumulates unevaluated operators on the outside and evaluates them after the base is reached. The accumulating form evaluates operators as they accumulate; when the base is reached, the result is returned.

Mathematical: if (base-case isTrue) then base-value else F[recursive-step]

Accumulating: if (base-case isTrue) then accum else F[recursive-step, accum+step]

Primitive Recursion Templates

These templates refer to arbitrary functions `F` on types of data structures. `P` is an arbitrary Boolean function, `G` and `H` are arbitrary domain functions

General:
```
F[x] = if P[x] then G[x]
             else H[F[x]]
```

Unstructured:
```
F[x] = if P[x] then G[x]
             else F[H[x]]
```

Integers:
```
F[n] = if n=0 then G[0]
             else H[n-1,F[n-1]]
```

Lists:
```
F[u] = if null[u] then G[nil]
             else H[head[u],tail[u],F[tail[u]]]
```

Trees:
```
F[v] = if leaf[v] then G[v]
             else H[left[v],right[v],F[left[v]],F[right[v]]]
```

Graphs:
```
F[v] = if null[v] then G[nil]
             else H[adjacent[v],F[not-visited[adjacent[v]]]]
```

Some Programming Examples

Here are a variety of programming styles for the function Factorial. Note that as the style moves more toward mathematical and away from procedural, the code becomes more succinct. It also becomes easier to debug and to verify. In general, the evolution of programming languages is away from CPU specifics and toward mathematical generalizations. This evolution simply means that more and more of the low-level mechanical details are moved out of the programming language and into the compiler.

```
proceduralFactorial[n_Integer?Positive] :=
      Block[{iterator = n, result = 1},
            While[iterator != 1,
                  result = result * iterator;
                  iterator = iterator - 1];
            result]


sugaredProceduralFactorial[n_] :=
      Block[{result = 1},
            Do[result = result*i,{i,1,n}];
            result]


recursiveFactorial[n_] :=
      If[n == 1,1,n*recursiveFactorial[n-1]]


rulebasedFactorial[1] = 1;
rulebasedFactorial[n_] := n*rulebasedFactorial[n-1]


mathematicalFactorial[n_] :=
      Apply[Times,Range[n]]
```

Program Verification, Induction Exercises

Here is a collection of relatively simple program transformations for practice of inductive proof and recursive implementation. Domain definitions and facts are provided. Use simple numerical and algebraic facts in the case of integers.

There are several approaches you might take to this exercise (in order of difficulty):

1. Use the domain theories and the induction principle to write recursive code.
2. Use the domain axioms to prove the assertions algebraically by hand.
3. Submit a subset of the axioms and rules to an algorithmic theorem prover (such as Mathematica, Maple, Reduce, Otter, or any of the tools available on the web).

Induction in the Integer Domain:

`{i,j,k,n}` are positive integers. Note that all integer functions can be defined inductively.

Definitions

```
i + j  =def=  i + 0 = i
              i + next[j] = (i + j) + 1

i * j  =def=  i * 0 = 0
              i * next[j] = (i * j) + i

i ^ j  =def=  i ^ 0 = 1
              i ^ next[j] = (i ^ j) * i

sum[n] =def=  sum[0] = 0
              sum[i+1] = sum[i] + (i + 1)

fac[n] =def=  fac[0] = 1
              fac[i+1] = fac[i] * (i + 1)

fib[n] =def=  fib[1] = fib[2] = 1
              fib[i+2] = fib[i+1] + fib[n]
```

Prove

- `(i * i) = (i ^ 2)`
- `(i * j) + (i * k) = i * (j + k)`
- `(2 * sum[n]) = n * (n + 1)`
- `(n ^ 2) = (2 * sum[n-1]) + n`
- `(3 * sum[n^2]) = (2*n + 1) * sum[n]`
- `fib[n^2] = fib[n+1] * fib[n]`
- `sum[n^3] = (sum[n] ^ 2)`

Recursive Programming

- Write recursive programs for the following functions:

	Domain
equal[a,b]	All domains -> Boolean
difference[i,j]	Natural numbers -> Integer
fibonacci[i]	Natural numbers -> Natnum
accumulating-fibonacci[i]	Natural numbers -> Natnum
greater-than[i,j]	Integers -> Boolean
remainder[i,j]	Integers -> Natnum
substitute[x,y,z]	Lists -> List
ordered-insert[item,x]	List -> List
same-length[x,y]	Strings -> Boolean (do not use Integers)
remove-substring[x,y]	Strings -> String
explode[x]	String -> List
fill[item,a]	Array
ordered[a]	Array -> Boolean
is-path[list,x]	Trees -> Boolean
depth[x]	Trees -> Integer
flatten[x]	Trees -> List
search[x, property]	Trees -> node
best-leaf[x]	Trees -> atom
occurrences[value,x]	Trees -> Integer
balanced-branches[x]	Trees -> Boolean
union[x,y]	Sets -> Set
intersection[x,y]	Sets -> Set
connected[n1,n2,g]	Graphs -> Boolean
delete-vertex[v,g]	Graphs -> graph
complement[g]	Graph -> Graph
reachable[v1,v2,g]	Graph -> Boolean
partition[x]	List -> Lists (harder)
tautology[lf]	Boolean -> Boolean (harder)

- Now *prove* that each program is correct.

- Do these recursive programs *terminate*?

Oscillate, for one integer (hint: try n=27):

```
osc[n] =def=
      if n=1 then 1
            else if even[n] then osc[n/2]
                        else osc[3n+1]
```

Ackerman, for two integers m and n:

```
ack[m,n] =def=
      if m=0 then n+1
            else if n=0 then ack[m-1,n]
                        else ack[m-1, ack[m, n-1]]
```

Ackerman, for two strings:

```
sack[x,y] =def=
      if (char x) then x·y
            else if (char y) then sack[rest[x],y]
                        else sack[rest[x], sack[x, rest[y]]
```

Takeuchi for three integers:

```
tak[i,j,k] =def=
      if i<=j then k
            else tak[tak[i-1,j,k],tak[j-1,k,i],tak[k-1,i,j]]
```

- How would you prove the correctness of this do-loop?

```
(2 ^ i) = (res := 1; for n from 1 to i do (res := res * 2))
```

Induction in the String Domain:

`{u,v}` are characters, `{x,y}` are strings

- Prove that the length of two strings concatenated together is the sum of the lengths of each.
- Prove that the length of the reverse of a string is the same as the length of the string.
- Develop a theory of **substrings** of a string. Here are the axioms you'll need:

The definition of a substring:

```
x sub y  =def=  z1*x*z2 = y
```

The empty string is a substring of every string

```
E sub y
```

No string is a substring of the empty string

```
not(y sub E)
```

Prefixing a character to a string maintains the substring relation

```
if (x sub y) then (x sub u·y)
```

The following three properties of the substring relation establish that *substring is an ordering relation.*

transitivity	if `s1` is a substring of `s2`, and `s2` is a substring of `s3`, then `s1` is a substring of `s3`
antisymmetry	if two strings are substrings of each other, they are equal
reflexivity	a string is a substring of itself

- Prove or define the above relations. Then prove:
  - A string is a substring of itself when a character is prefixed.
  - A string is a substring of the empty string when it is the empty string.
  - Substring implies all the characters in the substring are in the string.
  - The length of a substring is equal to or less than the length of the string.

- Extend the results:

  The definition of a **proper** substring:

  ```
  x proper-sub y  =def=  not(z1=E and z2=E) and z1*x*z2 = y
  ```

  - Prove the properties of proper substrings (transitivity, irreflexivity, asymmetry)

- Use the results:

  The definition of a palindrome:

  ```
  palin[x]  =def=  (x = rev[x])
  ```

  - Write a provable implementation of a palindrome tester/generator

  ```
  palin[x] = (x = y*rev[y]) or (x = y*u*rev[y])
  ```

Finally, here's an example of how a larger provable program might be built to construct long palindrome sentences, using a dictionary and a grammar checker:

1. select an arbitrary word from the dictionary
2. find the words which begin with the reverse of that word.
3. identify the end substring of the new word which is not covered by the first word
4. reverse the uncovered substring and find words ending with that substring
5. recur to step 2 until failure to match
6. use the grammar parser to screen the result for proper sentence structure
7. recur to step 1 to try again