ICOM 4075:

Foundations of Computing

Lecture 9:

Construction Techniques (2)

Department of Electrical and Computer Engineering
University of Puerto Rico at Mayagüez
Summer 2005

Lecture Notes Originally Written By Prof. Yi Qian

Homework 8 (Sugeridos para examen)

Section 3.1: (pp.142-145)

1. b.
2. b. d. f.
3. b.
4. b. d.
6. b. d. f. h. j.
7. b. d.
9. b.
10. b. d. f. h.
11. b.
12.
14.
16. b.
17. b.
18. b. d.
21.

Reading

- Textbook: James L. Hein, *Discrete Structures, Logic, and Computability*, 2nd edition, Chapter 3. Section 3.2

Recursive Functions and Procedures

- *Procedure*
  - From a computer science point of view a *procedure* is a program that performs one or more actions. So there is no requirement to return a specific value.
    - E.g., the execution of a statement like *print(x, y)* will cause the values of x and y to be printed. In this case, two actions are performed, and no values are returned.
    - A procedure may also return one or more values through its argument list. E.g., a statement like *allocate(m, a, b)* might perform the action of allocating a block of *m* memory cells and return the values *a* and *s*, where *a* is the beginning address of the block and *s* tells whether the allocation was successful.

Definition of Recursively Defined

- A function or a procedure is said to be *recursively defined* if it is defined in terms of itself.

  i.e., a function f is recursively defined if at least one value f(x) is defined in terms of another value f(y), where $x \neq y$.

  Similarly, a procedure P is recursively defined if the actions of P for some argument x are defined in terms of the actions of P for another argument y, where $x \neq y$.
- Many useful recursively defined functions have domains that are inductively defined sets. Similarly, many recursively defined procedures process elements from inductively defined sets.
- Constructing a Recursively Defined Function

  If S is an inductively defined set, then we can construct a function f with domain S as follows:
  - 1. For each basis element $x \in S$, specify a value for f(x).
  - 2. Give rules that, for any inductively defined element $x \in S$, will define f(x) in terms of previously defined values of f.
- Constructing a Recursively Defined Procedure

  If S is an inductively defined set, we can construct a procedure P to process the elements of S as follows:
  - 1. For each basis element $x \in S$, specify a set of actions for P(x).
  - 2. Give rules that, for any inductively defined element $x \in S$, will define the actions of P(x) in terms of previously defined actions of P.

Numbers

- To calculate the sum of the first n natural numbers for any $n \in N$.

  Let f(n) denote the desired sum, f(n) = 0 + 1 + 2 + … + n.

  We observe that f(0) = 0, f(1) = 1, f(2) = f(1) + 2 = 3, f(3) = f(2) + 3 = 6, f(4) = f(3) + 4 = 10, and so on.

  When n > 0, the definition can be transformed in the following way:

  f(n) = 0 + 1 + 2 + … + n = (0 + 1 + 2 + … + (n – 1)) + n = f(n – 1) + n.

  This gives the recursive part of a definition of f for any n > 0. For the basis case we have f(0) = 0. So we can write the following recursive definition for f:

  $$f(0) = 0,$$
  $$f(n) = f(n-1) + n \quad \text{for } n > 0.$$

- There are two alternative forms that can be used to write a recursive definition.
  - One form expresses the definition as an *if-then-else* equation. For example, f can be described in the following way:

    f(n) = if n = 0 then 0 else f(n – 1) + n.
  - Another form expresses the definition as equations whose left sides determine which equation to use in the evaluation of an expression rather than a conditional like n > 0. Such a form is called a *pattern-matching* definition because the equation chosen to evaluate f(x) is determined uniquely by which left side f(x) matches.

    For example, f can be described in the following way:

    $$f(0) = 0$$
    $$f(n + 1) = f(n) + n + 1.$$
  - A recursively defined function can be evaluated by a technique called *unfolding* the definition. E.g., f(4) = … …

Using the Floor Function

- Let f: N → N be defined in terms of the floor function as follows:

  f(0) = 0,

  f(n) = f(floor(n/2)) + n   for n > 0.

  Notice in this case that f(n) is not defined in terms of f(n – 1) but rather in terms of f(floor(n/2)).

  For example, f(16) = f(8) + 16.

  The first few values are f(0) = 0, f(1) = 1, f(2) = 3, f(3) = 4, and f(4) = 7.

  To calculate f(25):

$$\begin{aligned} f(25) &= f(12) + 25 \\ &= f(6) + 12 + 25 \\ &= f(3) + 6 + 12 + 25 \\ &= f(1) + 3 + 6 + 12 + 25 \\ &= f(0) + 1 + 3 + 6 + 12 + 25 \\ &= 0 + 1 + 3 + 6 + 12 + 25 \\ &= 47 \end{aligned}$$

Adding Odd Numbers

- Let $f: N \to N$ denote the function to add up the first n odd natural numbers. So f has the following informal definition:

  $$f(n) = 1 + 3 + \ldots + (2n + 1).$$

  For example, the definition tells that $f(0) = 1$. For $n > 0$ we can make the following transformation of $f(n)$ into an expression in terms of $f(n - 1)$:

  $$\begin{aligned} f(n) &= 1 + 3 + \ldots + (2n + 1) \\ &= (1 + 3 + \ldots + (2n - 1)) + (2n + 1) \\ &= (1 + 3 + \ldots + 2(n - 1) + 1) + (2n + 1) \\ &= f(n - 1) + 2n + 1. \end{aligned}$$

  So we can make the following recursive definition of f:

  $$\begin{aligned} &f(0) = 1, \\ &f(n) = f(n - 1) + 2n + 1 \qquad \text{for } n > 0. \end{aligned}$$

  Alternatively, we can write the recursive part of the definition as

  $$f(n + 1) = f(n) + 2n + 3.$$

  We can also write the definition in the following *if-then-else* form:

  $$f(n) = \text{if } n = 0 \text{ then } 1 \text{ else } f(n - 1) + 2n + 1.$$

  Here is the evaluation of f(3) using the *if-then-else* definition:

  $$\begin{aligned} f(3) &= f(2) + 2(3) + 1 \\ &= f(1) + 2(2) + 1 + 2(3) + 1 \\ &= f(0) + 2(1) + 1 + 2(2) + 1 + 2(3) + 1 \\ &= 1 + 2(1) + 1 + 2(2) + 1 + 2(3) + 1 \\ &= 1 + 3 + 5 + 7 \\ &= 16 \end{aligned}$$

The Rabbit Problem

- The Fibonacci numbers are the numbers in the sequence

  0, 1, 1, 2, 3, 5, 8, 13, …

  where each number after the first two is computed by adding the preceding two numbers.
- Starting with a pair of rabbits, how many pairs of rabbits can be produced from that pair in a year if it is assumed that every month each pair produces a new pair that becomes productive after one month?

  For example, if we don't count the original pair and assume that the original pair needs one month to mature and that no rabbits die, then the number of new pairs produced each month for 12 consecutive months is given by the sequence

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.

  The sum of these numbers, which is 232, is the number of pairs of rabbits produced in one year from the original pair.
- Fibonacci numbers seem to occur naturally in many unrelated problems. They can be defined recursively: letting *fib(n)* be the *n*th Fibonacci number, we can define *fib* recursively as follows:

  $fib(0) = 0$,

  $fib(1) = 1$,

  $fib(n) = fib(n - 1) + fib(n - 2)$ for $n \geq 2$.

  The third line could be written in pattern matching form as

  $fib(n + 2) = fib(n + 1) + fib(n)$.

  The definition of fib in *if-then-else* form looks like

```
fib(n) = if n = 0 then 0
         else if n = 1 then 1
         else fib(n – 1) + fib(n – 2).
```

Sum and Product Notation

- Many definitions and properties that we use without thinking are recursively defined.
- For example, given a sequence of numbers (a_1, a_2, …, a_n) we can represent the sum of the sequence with *summation notation* using the symbol Σ as follows.

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n$$

This notation has the following recursive definition, which makes the practical assumption that an empty sum is 0.

$$\sum_{i=1}^{n} a_i = \text{if } n = 0 \text{ then } 0 \text{ else } a_n + \sum_{i=1}^{n-1} a_i$$

Similarly we can represent the product of the sequence with the following *product notation*, where the practical assumption is that an empty product is 1.

$$\prod_{i=1}^{n} a_i = \text{if } n = 0 \text{ then } 1 \text{ else } a_n \cdot \prod_{i=1}^{n-1} a_i$$

In the special case where (a_1, a_2, …, a_n) = (1, 2, …, n) the product defines popular *factorial function*, which is denoted by n! and is read "n factorial". In other words, we have

$$n! = (1)(2)\cdots(n - 1)(n).$$

For example, 4! = 4·3·2·1 = 24, and 0! = 1. So we can define n! in the following recursive form.

$$n! = \text{if } n = 0 \text{ then } 1 \text{ else } n\cdot(n - 1)!$$

Strings

- To calculate the complement of any string over the alphabet {a, b}. For example, the complement of the string *bbab* is *aaba*.

  Let f(x) be the complement of x. To find a recursive definition for f we'll start by observing that an arbitrary string over {a, b} is either Λ or has the form ay or by for some string y. So we'll define the result of f applied to each of these forms as follows:

$$\begin{aligned} f(\Lambda) &= \Lambda, \\ f(ax) &= bf(x), \\ f(bx) &= af(x). \end{aligned}$$

  For example, we'll evaluate f(bbab):

$$\begin{aligned} f(bbab) &= a\, f(bab) \\ &= aa\, f(ab) \\ &= aab\, f(b) \\ &= aaba. \end{aligned}$$

Prefix of Strings

- Consider the problem of finding the longest common prefix of two strings. A string *p* is a *prefix* of the string *x* if *x* can be written in the form *x* = *ps* for some string *s*.
  - For example, *aab* is the longest common prefix of the two strings *aabbab* and *aababb*.

- For two strings over {a, b}, let f(x, y) be the longest common prefix of x and y. To find a recursive definition for f we can start by observing that an arbitrary string over {a, b} is either the empty string Λ or has the form *as* or *bs* for some string *s*. So the string over {a, b} are an inductively defined set. We can define f(s, t) by making sure that we take care to define it for all combinations of *s* and *t*.

  Here is a definition of f in pattern-matching form:

  f(Λ, x) = Λ,
  f(x, Λ) = Λ,
  f(ax, by) = Λ,
  f(bx, ay) = Λ,
  f(ax, ay) = a f(x, y),
  f(bx, by) = b f(x, y).

  We can also put the definition in *if-then-else* form as follows:

```
f(s, t)  =  if s = Λ or t = Λ then Λ
            else if s = ax and t = ay then a f(x, y)
            else if s = bx and t = by then b f(x, y)
            else Λ.
```

  We'll demonstrate the definition of f by calculating f(aabbab, aababb):

```
f(aabbab, aababb) = a f(abbab, ababb)
                  = aa f(bbab, babb)
                  = aab f(bab, abb)
                  = aab Λ
                  = aab
```

Converting Natural Numbers to Binary

- From Section 2.1, we can represent a natural number x as

  x = 2(floor(x/2)) + x mod 2.

  This formula can be used to create a binary representation of x because x mod 2 is the rightmost bit of the representation. The next bit is found by computing floor(x/2) mod 2. The next bit is floor(floor(x/2)/2) mod 2, and so on.

  – E.g., compute the binary representation of 13.

  ```
  13 = 2 floor(13/2) + 13 mod 2 = 2(6) + 1
  6   = 2 floor(6/2) + 6 mod 2     = 2(3) + 0
  3   = 2 floor(3/2) + 3 mod 2     = 2(1) + 1
  1   = 2 floor(½)  + 1 mod 2      = 2(0) + 1
  ```

  We can read off the remainders in reverse order to obtain 1101, which is the binary representation of 13.

- To write a recursive definition for the function "binary" to compute the binary representation for a natural number. If x = 0 or x = 1, then x is its own binary representation. If x > 1, then the binary representation of x is that of floor(x/2) with the bit x mod 2 attached on the right end. So our recursive definition of binary can be written as follows, where "cat" is the string concatenation function.

  ```
  binary(0) = 0,
  binary(1) = 1,
  binary(x) = cat(binary(floor(x/2)), x mod 2)      for x > 1.
  ```

  The definition can be written in *if-then-else* form as

  ```
  binary(x) = if x = 0 or x = 1 then x
                  else cat(binary(floor(x/2)), x mod 2).
  ```

  For example:

  ```
  binary(13)  = cat(binary(6), 1) = cat(cat(binary(3), 0), 1) = cat(cat(cat(binary(1), 1), 0), 1)
              = cat(cat(cat(1, 1), 0), 1) = cat(cat(11, 0), 1) = cat(110, 1)
              = 1101
  ```

Lists

- Define the function f: N → lists(N) that computes the following backwards sequence:

  f(n) = <n, n-1, …, 1, 0>.

  With a little help from the cons function for lists, we can transform the informal definition of f(n) into a computable expression in terms of f(n-1):

  f(n) = <n, n-1, …, 1, 0>
  = cons(n, <n-1, …, 1, 0>)
  = cons(n, f(n-1)).

  Therefore, f can be defined recursively by

  f(0) = <0>.
  f(n) = cons(n, f(n-1)) for n > 0.

  This definition can be written in if-then-else form as

  f(n) = if n = 0 then <0> else cons(n, f(n-1)).

  To see how the evaluation works, look at the unfolding that results when we evaluate f(3):

  f(3) = cons(3, f(2))
  = cons(3, cons(2, f(1)))
  = cons(3, cons(2, cons(1, f(0))))
  = cons(3, cons(2, cons(1, <0>)))
  = cons(3, cons(2, <1, 0>))
  = cons(3, <2, 1, 0>)
  = <3, 2, 1, 0>.

  We haven't given a recursively defined procedure yet. So let's give one for the problem we've been discussing. For example, suppose that P(n) prints out the numbers in the list <n, n-1, …, 0>. A recursive definition of P can be written as follows.

```
P(n): if n = 0 then print(0)
      else
        print(n);
        P(n-1)
      fi.
```

Length of a List

- Let S be a set and let “length” be the function of type lists(S) → N, which returns the number of elements in a list. We can define length recursively by noticing that the length of an empty list is zero and the length of a nonempty list is one plus the length of its tail. A definition follows:

  length(< >) = 0,

  length(cons(x, t)) = 1 + length(t).

  Recall that the infix form of cons(x, t) is x :: t. So we could just as well write the second equation as

  length(x :: t) = 1 + length(t).

  Also, we could write the recursive part of the definition with a condition as follows:

  length(L) = 1 + length(tail(L))   for L ≠ < >.

  In if-then-else form the definition can be written as follows:

  length(L) = if L = < > then 0 else 1 + length(tail(L)).

  The length function can be evaluated by unfolding its definition. For example, we can evaluate length(<a, b, c>) as:

  length(<a, b, c>) = 1 + length(<b, c>)
  = 1 + 1 + length(<c>)
  = 1 + 1 + 1 + length(< >)
  = 1 + 1 + 1 + 0
  = 3.

The Distribution Function

- To write a recursive definition for the distribution function:

  dist(a, <b, c, d, e>) = <(a, b), (a, c), (a, d), (a, e)>.

  To discover the recursive part of the definition, we'll rewrite the example equation by splitting up the lists into head and tail components as follows:

  ```
  dist(a, <b, c, d, e>)  = <(a, b), (a, c), (a, d), (a, e)>
                         = (a, b) :: <(a, c), (a, d), (a, e)>
                         = (a, b) :: dist(a, <c, d, e>).
  ```

  That's the key to the recursive part of the definition. Since we are working with lists, the basis case is dist(a, < >), which we define as < >. So the recursive definition can be written as follows:

  ```
  dist(x, < >) = < >,
  dist(x, h :: T) = (x, h) :: dist(x, T).
  ```

  For example, we'll evaluate the expression dist(3, <10, 20>):

  ```
  dist(3, <10, 20> = (3, 10) :: dist(3, <20>)
                   = (3, 10) :: (3, 20) :: dist(3, < >)
                   = (3, 10) :: (3, 20) :: < >
                   = (3, 10) :: <(3, 20)>
                   = <(3, 10), (3, 20)>.
  ```

  An if-then-else definition of dist takes the following form:

  ```
  dist(x, L) =  if L = < > then < >
                else (x, head(L)) :: dist(x, tail(L)).
  ```

The Pair Function

- The “pairs” function creates a list of corresponding elements from two lists. For example,

  pairs(<a, b, c>, <1, 2, 3>) = <(a, 1), (b, 2), (c, 3)>.

  To discover the recursive part of the definition, we’ll rewrite the example equation by splitting up the lists into head and tail components as follows:

  ```
  pairs(<a, b, c>, <1, 2, 3>) = <(a, 1), (b, 2), (c, 3)>
                              = (a, 1) :: <(b, 2), (c, 3)>
                              = (a, 1) :: pairs(<b, c>, <2, 3>).
  ```

  Now the pairs function can be recursively by the following equations:

  pairs(< >, < >) = < >,

  pairs(x :: T, y :: U) = (x, y) :: pairs(T, U).

  For example, we’ll evaluate the expression pairs(<a, b>, <1, 2>):

  ```
  pairs(<a, b>, <1, 2>) = (a, 1) :: pairs(<b>, <2>)
                        = (a, 1) :: (b, 2) :: pairs(< >, < >)
                        = (a, 1) :: (b, 2) :: < >
                        = (a, 1) :: <(b, 2)>
                        = <(a, 1), (b, 2)>.
  ```

The ConsRight Function

- Suppose we need to give a recursive definition for the sequence function. Recall, for example, that seq(4) = <0, 1, 2, 3, 4>. In this case, good old function "cons" doesn't seem up to the task. For example, if we somehow have computed seq(3), then cons(4, seq(3)) = <4, 0, 1, 2, 3>. It would be nice if we had a constructor to place an element on the right of a list, just as cons places an element on the left of a list. We'll write a definition "consR" to do just that. For example, we want

  consR(<a, b, c>, d) = <a, b, c, d>.

  We can get an idea of how to proceed by rewriting the above equation as follows in terms of the infix form of cons:

  ```
  consR(<a, b, c>, d)  = <a, b, c, d>
                       = a :: <b, c, d>
                       = a :: consR(<b, c>, d).
  ```

  So the clue is to split the list <a, b, c> into its head and tail. We can write the recursive definition of consR using the if-then-else form as follows:

  ```
  consR(L, a) = if L = < > then <a>
                else head(L) :: consR(tail(L), a).
  ```

  This definition can be write in pattern-matching form as follows:

  ```
  consR(< >, a) = a :: < >,
  consR(b :: T, a) = b :: consR(T, a).
  ```

  For example, we can construct the list <x, y> with consR as follows:

  ```
  consR(consR(< >, x), y) = consR(x :: < >, y)
                          = x :: consR(< >, y)
                          = x :: y :: < >
                          = x :: <y>
                          = <x, y>.
  ```

Concatenation of Lists

- An important operation on lists is the concatenation of two lists into a single list. Let “cat” denote the concatenation function. Its type is lists(A) x lists(A) → lists(A). For example,

  cat(<a, b>, <c, d>) = <a, b, c, d>.

  Since both arguments are lists, we have some choices to make. Notice, for example, that we can rewrite the equation as follows:

  ```
  cat(<a, b>, <c, d>)   = <a, b, c, d>
                          = a :: <b, c, d>
                          = a :: cat(<b>, <c, d>)
  ```

  So the recursive part can be written in terms of the head and tail of the first argument list. Here’s an if-then-else definition for cat.

  ```
  cat(L, M) = if L = < > then M
               else head(L) :: cat(tail(L), M).
  ```

  We’ll unfold the definition to evaluate the expression cat(<a, b>, <c, d>):

  ```
  cat(<a, b>, <c, d>)   = a :: cat(<b>, <c, d>)
                          = a :: b :: cat(< >, <c, d>)
                          = a :: b :: <c, d>
                          = a :: <b, c, d>
                          = <a, b, c, d>.
  ```

  We can also write cat as a recursively defined procedure that prints out the elements of the two lists:

  ```
  cat(K, L):    if K = < > then print(L)
                 else
                    print(head(K));
                    cat(tail(K), L)
                 fi.
  ```

Sorting a Lists by Insertion

- To define a function to sort a list of numbers by repeatedly inserting a new number into an already sorted list of numbers. Suppose “insert” is a function that does this job. Then the sort function itself is easy. For a basis case, notice that the empty list is already sorted. For the recursive case we sort the list x :: L by inserting x into the list obtained by sorting L. The definition can be written as follows:

  sort(< >) = < >,

  sort(x :: L) = insert(x, sort(L)).

  Everything seems to make sense as long as insert does its job. We’ll assume that whenever the number to be inserted is already in the list, then a new copy will be placed to the left of the one already there. Now let’s define insert. Again, the basis case is easy. The empty list is sorted, and to insert x into < >, we simply create the singleton list <x>. Otherwise – if the sorted list is not empty – either x belong on the left or the list, or it should actually be inserted somewhere else in the list. An if-then-else definition can be written as follows:

```
insert(x, S) = if S = < > then <x>
                   else if x ≤ head(S) then x :: S
                   else head(S) :: insert(x, tail(S)).
```

  Notice that insert works only when S is already sorted. For example, we’ll unfold the definition of insert(3, <1, 2, 6, 8>):

```
insert(3, <1, 2, 6, 8>) = 1 :: insert(3, <2, 6, 8>)
                        = 1 :: 2 :: insert(3, <6, 8>)
                        = 1 :: 2 :: 3 :: <6, 8>
                        = <1, 2, 3, 6, 8>.
```

The Map Function

- We'll construct a recursive definition for the map function. Recall, for example that

  map(f, <a, b, c>) = <f(a), f(b), f(c)>.

  Since the second argument is a list, it makes sense to define the basis case as map(f, < >) = < >. To discover the recursive part of the definition, we'll rewrite the example equation as follows:

  ```
  map(f, <a, b, c>) = <f(a), f(b), f(c)>
                    = f(a) :: <f(b), f(c)>
                    = f(a) :: map(f, <b, c>).
  ```

  So the recursive part can be written in terms of the head and tail of the input list. Here's an if-then-else definition for map.

  ```
  map(f, L) = if L = < > then < >
              else f(head(L)) :: map(f, tail(L)).
  ```

  For example, we'll evaluate the expression map(f, <a, b, c>):

  ```
  map(f, <a, b, c>) = f(a) :: map(f, <b, c>)
                    = f(a) :: f(b) :: map(f, <c>)
                    = f(a) :: f(b) :: f(c) :: map(f, < >)
                    = f(a) :: f(b) :: f(c) :: < >
                    = <f(a), f(b), f(c)>.
  ```

Binary Trees

- To find some functions that computes properties of binary trees. To start, suppose we need to know the number of nodes in a binary tree. Since the set of binary trees over a particular set can be defined inductively, we should be able to come up with a recursively defined function that suits our needs. Let “nodes” be the function that returns the number of nodes in a binary tree. Since the empty tree has no nodes, we have nodes(< >) = 0. If the tree is not empty, then the number of nodes can be computed by adding 1 to the number of nodes in the left and right subtrees. The equational definition of nodes can be written as follows:

  nodes(< >) = 0,

  nodes(tree(L, a, R)) = 1 + nodes(L) + nodes(R).

  If we want the corresponding if-then-else form of the definition, it looks like

  ```
  nodes(T)  = if T = < > then 0
                  else 1 + nodes(left(T)) + nodes(right(T)).
  ```

  For example, we’ll evaluate nodes(T) for T = <<< >, a, < >>, b, < >>:

  ```
  nodes(T)  = 1 + nodes(<< >, a, < >>) + nodes(< >)
            = 1 + 1 + nodes(< >) + nodes(< >) + nodes(< >)
            = 1 + 1 + 0 + 0 + 0
            = 2.
  ```

A Binary Search Tree

- Suppose we have a binary search tree whose nodes are numbers, and we want to add a new number to the tree, under the assumption that the new tree is still a binary search tree. A function to do the job needs two arguments, a number x and a binary search tree T. Let the name of the function be “insert”.

  The basis case is easy. If T = < >, then return tree(< >, x, < >). The recursive part is straightforward. If x < root(T), then we need to replace the subtree left(T) by insert(x, left(T)). Otherwise, we replace right(T) by insert(x, right(T)). Notice that repeated elements are entered to the right. If we didn’t want to add repeated elements, then we could simply return T whenever x = root(T). The if-then-else form of the definition is

```
insert(x, T) = if T = < > then tree(< >, x, < >)
                   else if x < root(T) then
                           tree(insert(x, left(T)), root(T), right(T))
                   else
                           tree(left(T), root(T), insert(x, right(T))).
```

  Now suppose we want to build a binary search tree from a given list of numbers in which the numbers are in no particular order. We can use the insert function as the main ingredient in a recursive definition. Let “makeTree” be the name of the function. We’ll use two variables to describe the function, a binary search tree T and a list of numbers L.

```
makeTree(T, L) = if L = < > then T
                   else makeTree(insert(head(L), T), tail(L)).
```

  To construct a binary search tree with this function, we apply makeTree to pair of arguments(< >, L).

  The function makeTree can be defined another way. Suppose we consider the following definition for constructing an binary search tree:

```
makeTree(T, L) = if L = < > then T
                   else insert(head(L), makeTree(T, tail(L))).
```

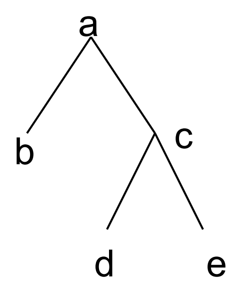
Traversing Binary Trees

- There're several useful ways to list the nodes of a binary tree. The three most popular methods of traversing a binary tree are called *preorder*, *inorder*, and *postorder*. We'll start with the definition of a preorder traversal.

- Preorder Traversal:

  The *preorder* traversal of a binary tree starts by visiting the root. Then there is a preorder traversal of the left subtree followed by a preorder traversal of the right subtree.

  For example, the preorder listing of the nodes of the binary tree in the following figure is <a, b, c, d, e>. It's common practice to write the listing without any punctuation symbols as a b c d e.



A Preorder Procedure

- Since binary trees are inductively defined, we can easily write a recursively defined procedure to output the preorder listing of a binary tree. For example, the following recursively defined procedure prints the preorder listing of its arguments T.

```
Preorder(T):  if T ≠ < > then
                  print(root(T));
                  Preorder(left(T));
                  Preorder(right(T))
              fi.
```

A Preorder Function

- To write a function to compute the preorder listing of a binary tree. Letting “preOrd” be the name of the preorder function, an equational definition can be written as follows:

  preOrd(< >) = < >,

  preOrd(tree(L, x, R)) = x :: cat(preOrd(L), preOrd(R)).

  The if-then-else form of preOrd can be written as follows:

  preOrd(T) = if T = < > then < >
  else root(T) :: cat(preOrd(left(T)), preOrd(right(T))).

  We can evaluate the expression preOrd(T) for the tree T = <<< >, a, < >>, b, < >>:

  preOrd(T) = b :: cat(preOrd(<< >, a, < >>), preOrd(< >))
  = b :: cat(a :: cat(preOrd(< >), preOrd(< >)), preOrd(< >))
  = b :: cat(a :: < >, < >)
  = b :: cat(<a>, < >)
  = b :: <a>
  = <b, a>.

The Repeated Element Problem

- To remove repeated elements from a list: Depending on how we proceed, there might be different solutions.
- For example, we can remove the repeated elements from the list <u, g, u, h, u> in three ways, depending on which occurrence of u we keep: <u, g, h>, <g, u, h>, or <g, h, u>. We'll solve the problem by always keeping the leftmost occurrence of each element. Let "remove" be the function that takes a list L and returns the list remove(L), which has no repeated elements and contains the leftmost occurance of each element of L.
- To start things off, we can say remove(< >) = < >. Now if L ≠ < >, then L has the form L = b :: M for some list M. In this case, the head of remove(L) should be b. The tail of remove(L) can be obtained by removing all occurrences of b from M and then removing all repeated elements from the resulting list. So we need a new function to remove all occurrences of an element from a list.
- Let removeAll(b, M) denote the list obtained from M by removing all occurrences of b. Now we can write an equational definition for the remove function as follows:

  remove(< >) = < >,

  remove(b :: M) = b :: remove(removeAll(b, M)).

  we can rewrite the solution in if-then-else form as follows:

```
remove(L) = if L = < > then < >
            else head(L) :: remove(removeAll(head(L), tail(L))).
```

The Repeated Element Problem (cont.)

- To complete the task, we need to define the "removeAll" function. The basis case is removeAll(b, < >) = < >. If M ≠ < >, then the value of removeAll(b, M) depends on head(M). If head(M) = b, then throw it away and return the value of removeAll(b, tail(M)). But if head(M) ≠ b, then it's a keeper. So we should return the value head(M) :: removeAll(b, tail(M)). We can write the definition in if-then-else form as follows:

```
removeAll(b, M) = if M = < > then < >
                  else if head(M) = b then
                    removeAll(b, tail(M))
                  else
                    head(M) :: removeAll(b, tail(M)).
```

- As an example, we can evaluate the expression removeAll(b, <a, b, c, b>):

```
removeAll(b, <a, b, c, b>) = a :: removeAll(b, <b, c, b>)
                           = a :: removeAll(b, <c, b>)
                           = a :: c :: removeAll(b, <b>)
                           = a :: c :: removeAll(b, < >)
                           = a :: c :: < >
                           = a :: <c>
                           = <a, c>.
```