**SE 2A04 Fall 2001**

# Recursion

Instructor: W. M. Farmer

Revised: 4 October 2001



## What is Recursion?

- **Recursion** is a method of defining something in terms of itself
  - One of the most fundamental ideas of computing
  - An alternative to iteration (loops)
  - Can make some programs easier to describe, write, and prove correct
- Both procedures and data structures can be defined by recursion
  - A set of procedures or data structures can be defined by **mutual recursion**



## Recursion Example: Factorial

```
MODULE Factorial;

IMPORT Out;

PROCEDURE FactRec*(x: LONGINT): LONGINT;
  (* Computes the factorial function using (nontail) recursion. *)
  BEGIN
    IF x < 0 THEN
      Out.String("Input must be nonnegative (ignore output).");
      RETURN -1;
    ELSIF x = 0 THEN
      RETURN 1;
    ELSE
      RETURN FactRec(x - 1) * x;
    END;
  END FactRec;

END Factorial.
```



## Semantics

- A recursive procedure can be understood as:
  - A definition of a procedure with an infinite body
  - An operational definition (little machine)
  - An implicit definition of a procedure that satisfies a certain property
- The use of recursion requires care and understanding
  - Recursive definitions can be nonsensical (i.e., nonterminating)
  - Sloppy use of recursion can lead to total confusion

## Implementation

- Recursive procedures are usually implemented using a **call stack**
  - The stack contains one **frame** per procedure call
  - The nesting depth of recursive calls does not need to be calculated before execution
- If the nesting depth of recursive calls is infinite, the procedure will run until the stack space is exhausted



## Quality Issues

- **Termination** is shown using a **well-founded ordering**
  - E.g., a strictly decreasing natural number value
- **Correctness** can be proved using induction
- **Efficiency**
  - In some cases, recursion can be highly inefficient in the use of space
  - In some cases, recursion can be executed in constant space



## Tail Recursion

- A procedure is **tail recursive** if nothing is left to do after each recursive call in the procedure body
- Tail recursive procedures can be made to execute in constant space:
  - In some programming languages, e.g., Scheme, the compiler ensures that tail recursive procedures execute in constant space
  - In other programming languages, tail recursive procedures can be redefined using iteration (which executes in constant space)



## Tail Recursion Example: Factorial

```
MODULE Factorial;

IMPORT Out;

PROCEDURE FactTailRec*(x: LONGINT): LONGINT;
  (* Computes the factorial function using tail recursion. *)
  BEGIN
    RETURN FactTailRecAux(x,1);
  END FactTailRec;

PROCEDURE FactTailRecAux(x: LONGINT; accum: LONGINT): LONGINT;
  BEGIN
    IF x < 0 THEN
      Out.String("Input must be nonnegative (ignore output).");
      RETURN -1;
    ELSIF x = 0 THEN
      RETURN accum;
    ELSE
      RETURN FactTailRecAux(x - 1, accum * x);
    END;
  END FactTailRecAux;

END Factorial.
```

## The BESTT Logic

- BESTT is a Basic Extended Simple Type Theory
  - Higher-order predicate logic
  - Includes type variables, lambda notation, and a definite description operator
  - Has support for reasoning with tuples, lists, and sets
- Several of the principal computer theorem systems are based on simple type theory: HOL, IMPS, PVS, TPS
- Reference:

  W. Farmer, "A Basic Extended Simple Type Theory", `http://imps.mcmaster.ca/wmfarmer/publications.html`



## Trees Example: Mathematical

- The set Tree of **binary integer trees** is defined recursively by the following statements:
  1. If $x \in \mathbf{Z}$, then $x \in$ Tree.
  2. If $x, y \in$ Tree, then $(x, y) \in$ Tree.
- The following tree functions are defined recursively as follows:

$$\forall t : \mathsf{Tree}\,.\ \mathsf{sum}(t) = \mathsf{if}(t \in \mathbf{Z},\ t,\ \mathsf{sum}(\mathsf{fst}(t)) + \mathsf{sum}(\mathsf{snd}(t)))$$

$$\forall t : \mathsf{Tree}\,.\ \mathsf{ht}(t) = \mathsf{if}(t \in \mathbf{Z},\ 1,\ 1 + \max(\mathsf{ht}(\mathsf{fst}(t)), \mathsf{ht}(\mathsf{snd}(t))))$$



## Trees Example: Planning Language 1

- Data Structure Type:

```
Tree = record
  atomic: Boolean;   (* true if integer; false if pair *)
  int:    Integer;
  left:   Tree;
  right:  Tree;
  end Tree;
```

- Constructors:

```
procedure MakeIntegerTree(i: Integer): Tree;
  return (true, i, null, null);
  end MakeIntegerTree;

procedure MakePairTree(t1,t2: Tree): Tree;
  return (false, null, t1, t2);
  end MakePairTree;
```



## Trees Example: Planning Language 2

- Selectors:

```
procedure IsIntegerTree(t: tree): Boolean;
  ((t.atomic  -> return true) |
   (~t.atomic -> return false))
  end IsIntegerTree;

procedure IsPairTree(t: tree): Boolean;
  return ~IsIntegerTree(t);
  end IsPairTree;

procedure GetInteger(t: tree): Integer;
  ((IsIntegerTree(t) -> return t.int) |
   (IsPairTree(t)    -> abort))
  end GetInteger;
```

# Trees Example: Planning Language 3

```
procedure GetLeftTree(t: tree): Tree;
  ((IsIntegerTree(t) -> abort) |
   (IsPairTree(t)    -> return t.left))
  end GetLeftTree;

procedure GetRightTree(t: tree): Tree;
  ((IsIntegerTree(t) -> abort) |
   (IsPairTree(t)    -> return t.right))
  end GetRightTree;
```



# Trees Example: Planning Language 4

- Mutators:

```
procedure SetInteger(t: tree; i: Integer);
  ((IsIntegerTree(t) -> t.int := i |
   (IsPairTree(t)    -> abort))
  end SetInteger;

procedure SetLeftTree(t: tree; t1: Tree);
  ((IsIntegerTree(t) -> abort) |
   (IsPairTree(t)    -> t.left := t1))
  end SetLeftTree;

procedure SetRightTree(t: tree; t2: Tree);
  ((IsIntegerTree(t) -> abort) |
   (IsPairTree(t)    -> t.right := t2))
  end SetRightTree;
```



# Trees Example: Planning Language 5

- Tree procedures:

```
procedure Sum(t: Tree): Integer;
  ((IsIntegerTree(t) -> return GetInteger(t)) |
   (IsPairTree(t)    -> return Sum(GetLeftTree(t)) +
                               Sum(GetRightTree(t)))
  end Sum;

procedure Height(t: Tree): Integer;
  ((IsIntegerTree(t) -> return 1) |
   (IsPairTree(t)    ->
      return 1 + Max(Height(GetLeftTree(t)),
                     Height(GetRightTree(t)))))
  end Height;
```



# Trees Example: Oberon 1

```
MODULE Trees;

IMPORT Out;

(* Data structure type: *)

TYPE

  Tree* = POINTER TO TreeRec;

  TreeRec = RECORD
    atomic: BOOLEAN;   (* True if integer; false if pair *)
    int:    INTEGER;
    left:   Tree;
    right:  Tree;
    END;
```

# Trees Example: Oberon 2

```
(* Constructors: *)

PROCEDURE MakeIntegerTree*(i: INTEGER): Tree;
  VAR t: Tree;
  BEGIN
    NEW(t);
    t^.atomic := TRUE;
    t^.int := i;
    t^.left := NIL;   (* Denotes null tree *)
    t^.right := NIL;  (* Denotes null tree *)
    RETURN t;
  END MakeIntegerTree;

PROCEDURE MakePairTree*(t1,t2: Tree): Tree;
  VAR t: Tree;
  BEGIN
    NEW(t);
    t^.atomic := FALSE;
    t^.int := 0;      (* Denotes null integer *)
    t^.left := t1;
    t^.right := t2;
    RETURN t;
  END MakePairTree;
```



# Trees Example: Oberon 3

```
(* Selectors: *)

PROCEDURE IsIntegerTree*(t: Tree): BOOLEAN;
  BEGIN
    RETURN t^.atomic;
  END IsIntegerTree;

PROCEDURE IsPairTree*(t: Tree): BOOLEAN;
  BEGIN
    RETURN ~IsIntegerTree(t);
  END IsPairTree;

PROCEDURE GetInteger*(t: Tree): INTEGER;
  BEGIN
    IF IsIntegerTree(t) THEN
      RETURN t^.int;
    ELSE
      Out.String("Error in Trees.GetInteger:
                  argument is a pair tree.");
      HALT(20);  (* Abort program *)
    END;
  END GetInteger;
```



# Trees Example: Oberon 4

```
PROCEDURE GetLeftTree*(t: Tree): Tree;
  BEGIN
    IF IsIntegerTree(t) THEN
      Out.String("Error in Trees.GetLeftTree:
                  argument is an integer tree.");
      HALT(20);  (* Abort program *)
    ELSE
      RETURN t^.left;
    END;
  END GetLeftTree;

PROCEDURE GetRightTree*(t: Tree): Tree;
  BEGIN
    IF IsIntegerTree(t) THEN
      Out.String("Error in Trees.GetRightTree:
                  argument is an integer tree.");
      HALT(20);  (* Abort program *)
    ELSE
      RETURN t^.right;
    END;
  END GetRightTree;
```



# Trees Example: Oberon 5

```
(* Mutators: *)

PROCEDURE SetInteger*(t: Tree; i: INTEGER);
  BEGIN
    IF IsIntegerTree(t) THEN
      t^.int := i;
    ELSE
      Out.String("Error in Trees.SetInteger:
                  argument is a pair tree.");
      HALT(20);  (* Abort program *)
    END;
  END SetInteger;

PROCEDURE SetLeftTree*(t: Tree; t1: Tree);
  BEGIN
    IF IsIntegerTree(t) THEN
      Out.String("Error in Trees.SetInteger:
                  argument is an integer tree.");
      HALT(20);  (* Abort program *)
    ELSE
      t^.left := t1;
    END;
  END SetLeftTree;
```

## Trees Example: Oberon 6

```
PROCEDURE SetRightTree*(t: Tree; t2: Tree);
  BEGIN
    IF IsIntegerTree(t) THEN
      Out.String("Error in Trees.SetInteger:
                  argument is an integer tree.");
      HALT(20);  (* Abort program *)
    ELSE
      t^.right := t2;
    END;
  END SetRightTree;

(* Tree Procedures: *)

PROCEDURE Sum*(t: Tree): INTEGER;
  BEGIN
    IF IsIntegerTree(t) THEN
      RETURN GetInteger(t);
    ELSE
      RETURN Sum(GetLeftTree(t)) + Sum(GetRightTree(t));
    END;
  END Sum;

END Trees.
```



## Pattern Matching: Problem

- Input:

  dat: Array $n$ of Character (data string)
  pat: Array $p$ of Character (pattern string)

- Output:

  mat: Array $n$ of Boolean (match array)

- Definition of match-at:

  match-at(pat, dat, $i$) $\equiv$
  $\forall j : \mathbf{N} .\ 0 \leq j < p \supset \text{pat}[j] = \text{dat}[i+j]$

- Specification $S$:

  $\forall i : \mathbf{N} .\ 0 \leq i < n \supset \text{mat}[i] \equiv \text{match-at}(\text{pat}, \text{dat}, i)$



## Pattern Matching: New Problem

- Input:

  dat: Array $n$ of Character (data string)
  pat: Array $p$ of Character (pattern string)
  mat: Array $n$ of Boolean (match array)
  $s$: **N** (start of potential match)
  $m$: **N** (number of matched characters)

- Output:

  mat: Array $n$ of Boolean (match array)

- Invariant $I$:

  $(m < p)\ \wedge$
  $(\forall i : \mathbf{N} .\ 0 \leq i < \min(s, n) \supset \text{mat}[i] \equiv \text{match-at}(\text{pat}, \text{dat}, i)) \wedge$
  $(\forall j : \mathbf{N} .\ 0 \leq j < m \supset \text{pat}[j] = \text{dat}[s+j])$



## Pattern Matching: match

```
procedure match(pat: Array p of Character;
                dat: Array n of Character): Array n of Boolean;
  var mat: Array n of Boolean;

  (* initialize mat *)
  var i : Integer;
  i := 0;
  it
    ((i < n -> mat[i] := false; go) |
     (i >= n -> stop))
  ti

  return match-aux(pat,dat,mat,0,0);
  end match;
```

# Pattern Matching: match-aux

```
procedure match-aux(pat: Array p of Character;
                    dat: Array n of Character;
                    mat: Array n of Boolean;
                    s: Natural;
                    m: Natural): Array n of Boolean;
  ((s + p > n -> return mat) |        (* dat is exhausted *)
   (s + p <= n ->
      (pat[m] = dat[s+m] ->           (* next char is matched *)
         m := m + 1;
         ((m = p ->                   (* pat is matched *)
             mat[s] := true;
             s := s + advance(pat,m);
             m := 0) |
          (~(m = p) -> skip))) |      (* pat is not (yet) matched *)
      (~(pat[m]=dat[s+m]) ->          (* next char is not matched *)
         (* mat[s] remains false *)
         s := s + advance(pat,m);
         m := 0)))
  return match-aux(pat,dat,mat,s,m)
  end match-aux;
```



# Pattern Matching: Conclusions

1. The recursion terminates since $(s, m)$ strictly increases in lexicographic order until $s + m \geq n$

2. $I$ is an invariant of the recursion:
   - $I$ is true when `match` calls `match-aux`
   - $I$ is true each time `match-aux` calls itself

   Hence, $I$ is true when `match-aux` terminates

3. Therefore, `match` satisfies $S$ (provided `advance` is correct)
   - Note: `advance` is correct, but not necessarily optimal, if it always returns 1