SE 2A04 Fall 2002

07 Recursion

Instructor: W. M. Farmer

Revised: 17 December 2002

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

PROCEDURE FactTailRecAux(x: INTEGER; accum: INTEGER): INTEGER;
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.
```

Example: MIS for Trees ADT (1)

- Imported modules: none required
- Interface:

```
INTERFACE Trees;
  TYPE Tree;
  PROCEDURE MakeLeaf(i: INTEGER): Tree;
  PROCEDURE MakePair(s,t: Tree): Tree;
  PROCEDURE IsLeaf(t: Tree): BOOLEAN;
  PROCEDURE GetInteger(t: Tree): INTEGER;
  PROCEDURE GetLeft(t: Tree): Tree;
  PROCEDURE GetRight(t: Tree): Tree;
  PROCEDURE Height(t: Tree): INTEGER;
  PROCEDURE Sum(t: Tree): INTEGER;
END Trees.
```

- Exceptions:
  - IntegerOfPair
  - SubtreeOfLeaf

Example: MIS for Trees ADT (2)

- Axioms:

  1. **Leaves are leaves.**
     $\forall\, t : \texttt{Tree} \;.\; \texttt{IsLeaf}(t) \Leftrightarrow \exists\, i : \texttt{INTEGER} \;.\; t = \texttt{MakeLeaf}(i)$

  2. **Pairs are never leaves.**
     $\forall\, s, t : \texttt{Trees} \;.\; \neg\texttt{IsLeaf}(\texttt{MakePair}(s, t))$

  3. **MakeLeaf is one-to-one.**
     $\forall\, i, j : \texttt{INTEGER} \;.\; \texttt{MakeLeaf}(i) = \texttt{MakeLeaf}(j) \Rightarrow i = j$

  4. **MarkPair is one-to-one.**
     $\forall\, s_1, s_2, t_1, t_2 : \texttt{Trees} \;.$
     $\texttt{MakePair}(s_1, t_1) = \texttt{MakePair}(s_1, t_1) \Rightarrow$
     $(s_1 = t_1 \wedge s_2 = t_2)$

Example: MIS for Trees ADT (3)

5. **Integer of a leaf.**
   $\forall\, i : \texttt{INTEGER} \,.\, \texttt{GetInteger}(\texttt{MakeLeaf}(i)) = i$

6. **Integer of a pair.**
   $\forall\, s, t : \texttt{Tree} \,.\, \texttt{GetInteger}(\texttt{MakePair}(s, t))\uparrow$
   [IntegerOfPair exception]

7. **Left of a pair tree.**
   $\forall\, s, t : \texttt{Tree} \,.\, \texttt{GetLeft}(\texttt{MakePair}(s, t)) = s$

8. **Left of a leaf.**
   $\forall\, i : \texttt{INTEGER} \,.\, \texttt{GetLeft}(\texttt{MakeLeaf}(i))\uparrow$
   [SubtreeOfLeaf exception]

Example: MIS for Trees ADT (4)

9. **Right of a pair tree.**
   $\forall\, s, t : \texttt{Tree} \,.\, \texttt{GetRight}(\texttt{MakePair}(s, t)) = t$

10. **Right of a leaf.**
    $\forall\, i : \texttt{INTEGER} \,.\, \texttt{GetRight}(\texttt{MakeLeaf}(i)) \uparrow$
    [SubtreeOfLeaf exception]

11. **Induction axiom for trees.**
    $\forall\, P : \texttt{Tree} \to \texttt{BOOLEAN} \,.$
    $[\forall\, i : \texttt{INTEGER} \,.\, P(\texttt{MakeLeaf}(i)) \wedge$
    $\forall\, s, t : \texttt{Tree} \,.\, P(s) \wedge P(t) \Rightarrow P(\texttt{MakePair}(s, t))]$
    $\Rightarrow \forall\, t : \texttt{Tree} \,.\, P(t)$

Example: MIS for Trees ADT (5)

12. **Height of a tree.**

$$\forall\, t : \texttt{Tree} \;.\; \texttt{Height}(t) =$$
$$\quad \textsf{if}(\texttt{IsLeaf}(t),$$
$$\quad\quad 1,$$
$$\quad\quad 1 + \textsf{if}(\texttt{Height}(\texttt{GetLeft}(t)) > \texttt{Height}(\texttt{GetRight}(t)),$$
$$\quad\quad\quad\quad \texttt{Height}(\texttt{GetLeft}(t)),$$
$$\quad\quad\quad\quad \texttt{Height}(\texttt{GetRight}(t))))$$

13. **Sum of a tree.**

$$\forall\, t : \texttt{Tree} \;.\; \texttt{Sum}(t) =$$
$$\quad \textsf{if}(\texttt{IsLeaf}(t),$$
$$\quad\quad \texttt{GetInteger}(t),$$
$$\quad\quad \texttt{Sum}(\texttt{GetLeft}(t)) + \texttt{Sum}(\texttt{GetLeft}(t)))$$

Trees Example: Oberon (1)

```
(*

Title: Binary Trees ADT

Interface:

INTERFACE Trees;
  TYPE Tree;
  PROCEDURE MakeLeaf(i: INTEGER): Tree;
  PROCEDURE MakePair(s,t: Tree): Tree;
  PROCEDURE IsLeaf(t: Tree): BOOLEAN;
  PROCEDURE GetInteger(t: Tree): INTEGER;
  PROCEDURE GetLeft(t: Tree): Tree;
  PROCEDURE GetRight(t: Tree): Tree;
  PROCEDURE Height(t: Tree): INTEGER;
  PROCEDURE Sum(t: Tree): INTEGER;
END Trees.

*)
```

Trees Example: Oberon (2)

```
MODULE Trees;

IMPORT Out;

(* Types *)

TYPE

  Tree* = POINTER TO TreeRec;

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

Trees Example: Oberon (3)

```
(* Exceptions: *)

PROCEDURE IntegerOfPairException();
BEGIN
  Out.String("Trees.IntegerOfPairException: A pair has no integer.");
  HALT(1)  (* Abort program *)
END IntegerOfPairException;

PROCEDURE SubtreeOfLeafException();
BEGIN
  Out.String("Trees.SubtreeOfLeafException: A leaf has not subtrees.");
  HALT(1)  (* Abort program *)
END SubtreeOfLeafException;
```

Trees Example: Oberon (4)

```
(* Constructors: *)

PROCEDURE MakeLeaf*(i: INTEGER): Tree;
  VAR t: Tree;
BEGIN
  NEW(t);
  t^.leaf := TRUE;
  t^.int := i;
  t^.left := NIL;
  t^.right := NIL;
  RETURN t
END MakeLeaf;

PROCEDURE MakePair*(t1,t2: Tree): Tree;
  VAR t: Tree;
BEGIN
  NEW(t);
  t^.leaf := FALSE;
  t^.int := 0;      (* This value will never be used *)
  t^.left := t1;
  t^.right := t2;
  RETURN t
END MakePair;
```

Trees Example: Oberon (5)

```
(* Selectors: *)

PROCEDURE IsLeaf*(t: Tree): BOOLEAN;
BEGIN
  RETURN t^.leaf
END IsLeaf;

PROCEDURE GetInteger*(t: Tree): INTEGER;
BEGIN
  IF IsLeaf(t) THEN
    RETURN t^.int
  ELSE
    IntegerOfPairException()
  END
END GetInteger;

PROCEDURE GetLeft*(t: Tree): Tree;
BEGIN
  IF Isleaf(t) THEN
    SubtreeOfLeafException()
  ELSE
    RETURN t^.left
  END
END GetLeft;
```

Trees Example: Oberon (6)

```
PROCEDURE GetRight*(t: Tree): Tree;
BEGIN
  IF IsLeaf(t) THEN
    SubtreeOfLeafException()
  ELSE
    RETURN t^.right
  END
END GetRight;

PROCEDURE Height*(t: Tree): INTEGER;
  VAR h1,h2: INTEGER;
BEGIN
  IF IsLeaf(t) THEN
    RETURN 1
  ELSE
    h1:= Height(GetLeft(t));
    h2:= Height(GetRight(t));
    IF h1 > h2 THEN
      RETURN 1 + h1
    ELSE
      RETURN 1 + h2
    END
  END
END Height;
```

Trees Example: Oberon (7)

```
PROCEDURE Sum*(t: Tree): INTEGER;
BEGIN
  IF IsLeaf(t) THEN
    RETURN GetInteger(t)
  ELSE
    RETURN Sum(GetLeft(t)) + Sum(GetRight(t))
  END
END Sum;

END Trees.
```