CS 2SC3 and SE 2S03 Fall 2008

# 06 Advanced Procedures

William M. Farmer

Department of Computing and Software
McMaster University

19 November 2008

# Higher-Order Procedures

- A higher-order function is a function that either takes functions as input or returns functions as output.
- A higher-order procedure is a procedure that represents a higher-order function.
- Higher-order functions are directly represented in OCaml.
- Higher-order functions are represented in C using function pointers, i.e., pointers that point to the address of a function.
- Higher-order procedures are invaluable for building complex procedures from simpler procedures.

# Function Pointers in C

- A function name is bound to the starting address in memory of the code that implements the function.
- A function pointer is a variable that holds the address of a function.
- Function pointers are used to indirectly store functions and to pass functions to other functions as input and output values.
- The syntax for declaring a function pointer is:

  $t$ (*fun_ptr)($t_1$ $p_1, \ldots, t_n$ $p_n$);

  Note: The parameter names $p_1, \ldots, p_n$ are optional.
- The syntax for applying the function that a function pointer references is:

  (*fun_ptr)($a_1, \ldots, a_n$)

# Polymorphic Procedures

- A procedure is polymorphic if it can be applied to different types.
- In OCaml, polymorphic procedures are defined automatically when input and output types are not fully specified.
  - The execution of polymorphic procedures in OCaml is type safe.
- In C, polymorphic procedures are defined using the `void *` type.
  - The `void *` acts as a universal type.
  - The execution of polymorphic procedures in C is not type safe.
- The use of polymorphic procedures allows code to be more generic, more powerful.

# 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.
- 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.
  - Correctness is proved by induction.

# Semantics of Recursive Procedures

A recursive procedure can be understood as:

1. Declarative definition: A definition of a function with an infinite body.
2. Operational definition: A definition of a special-purpose computer.
3. Fixed point definition: An implicit definition of a function $f$ that satisfies an equation of the form $f = H(f)$.

# Implementation of Recursive Procedures

- Recursive procedures are usually implemented using the call stack.
  - The stack contains one frame for each call of the recursive procedure.
  - 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.
  - For example, 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 (e.g., in standard implementations of C).
  - In some cases, recursion can be executed in constant space (e.g., with tail recursive procedures in Scheme or OCaml).

# 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 and OCaml, 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).
- Loops can be replaced with the use of tail recursion.