COMPSCI 1JC3

Introduction to Computational Thinking

Fall 2017

10 Three Problem Solving Methods

William M. Farmer

Department of Computing and Software
McMaster University

November 13, 2017

Admin

- Midterm 1 will be held on Friday at 19:00–21:00 pm.
  - Testing rooms:

    MDCL 1102 (students Aksamit to Khanna).
    MDCL 1105 (students Lenko to Zhou).
  - 30 multiple choice questions.
    - Covers everything up to the end of Week 09.
    - Will be electronically marked.
    - Bring some HB pencils with you.
  - Two-stage format.
- Discussion sessions this week:
  - Wednesday: Regular discussion session.
  - Thursday: Review session for Midterm Test 2.
- Office hours: To see me please send me a note with times.
- Are there any questions?

Advice

- Focus on learning concepts, not memorizing answers to questions!
  - Understanding concepts prepares you for many questions.
  - Memorizing the answer to one question will prepare you for just one question.
- Organize what you have learned!
  - Divide and conquer the material.
  - Take notes and then organize them into an outline of the course.
  - Teach the material to others.

Review

1. Why information security is unique.
2. Confidentiality, integrity, availability.
3. Conventional encryption.
4. Public-key encryption.
5. Hash functions.
6. Login authentication.

Problem Solving Methods

- There are many good methods for solving problems.
- Three of my favorite problem solving methods are:
  1. Recursion and induction.
  2. Little languages.
  3. Copy, modify, compare, and generalize (CMCG).

Recursion and Induction

- Recursion is a method of defining a structure (i.e., a structured set of values) or a function in terms of itself.
  - One of the most fundamental ideas of computing.
  - Can make specifications, descriptions, and programs easier to express, understand, and prove correct.
- Induction is a method of proof based on a recursively defined structure.
  - The recursively defined structure and the proof method are specified by an induction principle.
  - Induction can be used to prove properties about recursively defined structures and functions.
- Recursion and induction are fundamental components of computational thinking!

Recursion (iClicker)

How comfortable are you now with defining functions by recursion?

A. Uncomfortable.
B. Slightly comfortable.
C. Very comfortable.
D. Extremely comfortable.

Example: Natural Numbers [1/2]

```
data Nat
  = Zero
  | Suc Nat
  deriving (Show)

natPlus :: Nat -> Nat -> Nat
x `natPlus` Zero    = x
x `natPlus` (Suc y) =
  Suc (x `natPlus` y)

natTimes :: Nat -> Nat -> Nat
x `natTimes` Zero    = Zero
x `natTimes` (Suc y) =
  x `natPlus` (x `natTimes` y)
```

Example: Natural Numbers [2/2]

- Induction principle for `Nat`: For any property `P`, if
  - `P Zero` holds and
  - `P (Suc x)` holds whenever `P x` holds,

  then `P x` holds for all values `x` of type `Nat`.
- Can be used to prove theorems about `Nat` and recursively defined functions on `Nat` such as `natPlus` and `natTimes`.
- Example theorems:

  Commutativity of `natPlus`:

  ```
  x `natPlus` y == y `natPlus` x.
  ```

  Commutativity of `natTimes`:

  ```
  x `natTimes` y == y `natTimes` x.
  ```


Little Languages

- What happens if you solve a problem but later the problem requirements change?
  - Your solution becomes a solution for the wrong problem.
  - You may need to start the problem solving process over.
- A better approach is to create a little language that can be used to solve a family of related problems.
  - The components of the language are designed to work together to solve a wide range of problems.
  - The family includes the problem at hand.
  - If the problem requirements change, the language can be used to construct a solution to the new problem.
  - This is called the little languages method.
- The little languages method is a fundamental component of computational thinking!

Little Languages Examples

- Differentiation rules.
  - The rules for symbolic differentiation form a language for computing derivatives.
- Computer graphics language.
  - The language has tools for creating a variety of graphical objects.
- Algebraic types.
  - The value constructors form a language for describing the members of the type.
- Software modules.
  - A well-designed software module provides a language that serves as an interface to module's implementation.

Copy, Modify, Compare, and Generalize

- The copy, modify, compare, and generalize (CMCG) method is a four-step design process to solve a problem P:
  1. Copy the solution S' to a related problem P'.
  2. Modify S' to make it a solution S for P.
  3. Compare S with S' to find mistakes in S' and to see if S' can be improved.
  4. Generalize S and S' to obtain a solution S^* that solves a family of problems that includes P and P'.
- CMCG trades short-term cost for long-term gain.

CMCG Example: Super Sigma [1/4]

```
sigmaPlus :: Num a =>
             Integer ->
             Integer ->
             (Integer -> a) ->
             a

sigmaPlus m n f
  | m > n    = 0
  | m <= n   =
      sigmaPlus m (n - 1) f + f n
```


CMCG Example: Super Sigma [2/4]

```
sigmaTimes :: Num a =>
              Integer ->
              Integer ->
              (Integer -> a) ->
              a

sigmaTimes m n f
  | m > n    = ∅ 1
  | m <= n   =
      sigmaTimes m (n - 1) f ≠ * f n
```


CMCG Example: Super Sigma [3/4]

```
sigmaAppend :: Integer ->
               Integer ->
               (Integer -> a String) ->
               a String

sigmaAppend m n f
  | m > n    = ∅ ""
  | m <= n   =
      sigmaAppend m (n - 1) f ≠ ++ f n
```


CMCG Example: Super Sigma [4/4]

```
import Data.Monoid

superSigma :: Monoid a =>
              Integer ->
              Integer ->
              (Integer -> a) ->
              a

superSigma m n f
  | m > n    = ∅ mempty
  | m <= n   =
      superSigma m (n - 1) f ≠ mappend f n
```