Course Overview

1. What is functional programming
2. The Haskell programming language
3. Some common data structures and algorithms
4. Combinators and its use in parsing
5. Theorem proving
6. Monadic I/O and a chat server example

Reading List

- Course web page: `www.cs.utoronto.ca/~trebla/fp/`
- Haskell and functional programming resources: `www.haskell.org`
- Haskell Tutorial: `www.haskell.org/tutorial/`
- Any good Haskell book, e.g.,
  - Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.
  - Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, second edition, 1998.
  - Any other books suggested on the Haskell home page.

Traditional Programming

Factorial example in traditional programming:

```
int factorial(int n) {
  int p = 1, i = 1;
  while (i <= n) {
    p = p * i;
    i = i + 1;
  }
  return p;
}
```

- *Imperative*: program works by reading and writing *state variables*.
- Use loops.

Functional Programming

Factorial example in functional programming:

```
factorial n =
  let f p i = if i <= n then
                 f (p*i) (i+1)
              else p
  in f 1 1
```

- No state variables. Program works by passing parameters and returning values.
- Use recursion, most often tail recursion.

Functional Programming (cont.)

- Functions are first-class citizens like data value are:
  - can pass a function as a parameter
  - can create a new function on the fly

This function inputs a function f and outputs a function g with $g(x) = 2f(x)$

```
double f = let g x = 2 * f x in g
```

Example of use:

```
g = double factorial
twelve = g 3
```

Evaluation Policy

Suppose a function does not use an argument:

```
f x y = x
```

What will happen if we give a malicious parameter to the unused argument?

```
f 3 (1/0)
```

- Eager evaluation: parameters are evaluated always. All malicious parameters cause errors.
- Lazy evaluation: parameters are evaluated only when used. Unused parameters never cause errors.

The language of this course, Haskell, performs lazy evaluation.

Functional vs Imperative: Modes of Thinking

- Think recursion, not loop.
  - Base case, induction step.
  - Divide and conquer.
- No state variables.
  - If you really need them, make them arguments.
- Don't hesitate to pass functions as parameters and return functions.
  - In fact, the library is full of functions like this.

Functional vs Imperative: Pros and Cons

- Pros:
  - No side effects.
  - Easier to prove correct.
  - Shorter, higher level.
- Cons:
  - Harder to write.
  - The I/O model is harder to understand and use.
  - Slower.

The Haskell Language

- Named after the logician Haskell B. Curry.
- Summarizes a lot of mature ideas, research, and experience in functional programming.
- Purely functional. No side effects.
- Lazy evaluation.
- Strongly typed and polymorphic.

Expressions and Values

Expressions are things you want the computer to calculate.

```
3
3 + 4
factorial (3+4)
x + y
```

Values are the results of calculating expressions.

The values of the above expressions are, respectively:

```
3
7
5040
ERROR: Undefined variables
```

Types

Each expression and value has a *data type.*

Some typical types in Haskell:

- `Int`: machine-sized integer
- `Integer`: arbitrary size integer
- `[Integer]`: list of integers
- `Integer -> Integer`: function that maps an integer to an integer
- `Integer->Integer->Integer`: maps two integers to an integer
- `(Integer, Int)`: ordered pair of Integer and Int

Types (cont.)

Examples:

- `factorial` has type `Integer -> Integer`
- `factorial 3` has type `Integer`
- `[3, 4, 5]` has type `[Integer]`
- `(3, 4, 5)` has type `(Integer, Integer, Integer)`
- 3+4 has type `Integer`
- + has type `Integer->Integer->Integer`

Bindings/Definitions

We can *bind* an expression to an identifiers, i.e., *define* an identifier to be an expression.

```
ten = 1 + 2 + 3 + 4
```

Important: this does not create a state variable. We cannot change `ten` later.

More often, we bind functions (which are expressions) to function names.

```
square x = x * x
```

This says: here is an expression that is a function mapping x to $x \times x$. Bind this function to `square`.

Local Bindings

```
let x=3 in x*x
```

This is called a *let-expression*.

- The *body* of the expression is `x*x`.
- Within the scope of the expression, `x` is temporarily bound to `3`.
- Therefore, the value of the expression will be `9`.
- Outside the scope of the expression, the binding is invisible.

Local Bindings (cont.)

Multiple local bindings in a let-expression:

```
let x = 3
    y = x + 4
in x * y
```

Local bindings may also be used in a function expression:

```
fourth_power x = let x2 = x*x in x2*x2
factorial n =
  let f p i = if i <= n then f (p*i) (i+1) else p
  in f 1 1
```

Local Bindings for Definitions

There is a *where-clause* for local bindings in definitions.

```
seven = x + y where x = 3
                    y = x + 4

fourth_power x = x2*x2 where x2 = x*x

factorial n = f 1 1
  where f p i = if i <= n then f (p*i) (i+1) else p
```

Note: where-clauses are only for definitions, not expressions.

```
x*x where x=3   <--- wrong; use let instead
```