1 Reading / Source Material

- Scott, Chapter 10 (excluding 10.3, 10.4, 10.6)
- Hudak, "Conception, Evolution, and Application of Functional Programming Languages". http://dx.doi.org.libweb.lib.utsa.edu/10.1145/72551.72554

2 Objectives: Be able to …

1. define, compare, contrast, understand, and use vocabulary listed below
2. be able to describe what “pure” functional programming does not have
3. be able to describe what features make functional programming practical
4. be able to explain what key examples of functional programs do

Vocabulary

first-class function, higher-order function, implicit polymorphism, constructors (aggregates—not OO constructors, for structured objects), currying, referential transparent, equational reasoning

3 Outline

1. History
   (a) 1930’s, before general purpose computers
   (b) “effective procedure” = algorithm = constructive proof
   (c) formalized as automata (Turing Machine), symbolic manipulation, etc.
   (d) Church’s thesis = all are equally powerful
      i. computability (undecidable problems, halting problem)
      ii. Turing-complete
   (e) \( \lambda \)-calculus (\( \lambda \) = Greek letter “Lambda”)
   (f) Select, Influential Functional Languages
      ii. Backus’s FP (1978)
      iv. Haskell (1990)

2. “pure” functional programming
   (a) recursion and parameter passing, replaces iteration and assignment
   (b) lack of side-effects
(c) referential transparency  
(d) equational reasoning  

3. common functional language features:  
(a) expression-oriented, extensive recursion  
(b) first-class function values and higher-order functions  
(c) extensive polymorphism  
(d) expressions for complex data types  
   i. types  
      A. built-in list types and operators  
      B. user-defined algebraic (concrete) datatypes  
   ii. operations  
      A. structured function returns  
      B. constructors (aggregates) for structured objects  

4. lazy evaluation (esp. for pure functional languages)  

4 Examples  
(examples borrowed from Hudak)  

4.1 Recursion instead of mutation and loops  

IMPERATIVE  

n := x;  
a := 1;  
while n>0 do  
begn a := a*n;  
   n := n-1  
end;  

LISP  

(define fact(n)  
  (if (= n 0)  
    1  
    (* n (fact (- n 1))) ))  

HASKELL 1  

fact x  
where fact n  
    = if n==0 then 1  
    else n*fact(n-1)
HASKELL 2 (tail recursive)

\[
\text{fact}\ x\ 1 \\
\text{where}\ \text{fact}\ n\ a \\
\quad = \begin{cases} \\
\quad \text{if } n > 0 \text{ then } \text{fact} (n-1) (a*n) \\
\quad \text{else } a
\end{cases}
\]

4.2 Referential Transparency

\[
...\ x + x ... \\
\text{where } x = f\ a
\]

vs.

\[
...\ (f\ a) + (f\ a) ... 
\]

4.3 Higher-Order Functions, Lists, and Pattern Matching

LISP

\[
\text{(define mapcar (fun lst)} \\
\quad \text{(if (null lst)} \\
\quad\quad \text{nil} \\
\quad\quad \text{(cons (fun (car lst))} \\
\quad\quad\quad \text{(mapcar fun (cdr \text{1st}))})
\quad \text{))}
\]

\[
\text{(define f (x) (* 2 x))} \\
\text{(mapcar f (cons 1 (cons 2 nil)))}
\]

HASKELL

\[
\text{mapcar fun [] = []} \\
\text{mapcar fun (x:xs) = fun x : mapcar fun xs}
\]

4.4 Algebraic Datatypes

\[
\text{data List a} \\
\quad = \text{Nil} \mid \text{Cons a (List a)} \\
\text{data Tree b} \\
\quad = \text{Empty} \mid \text{Node b (List (Tree b))}
\]