Lecture 39:

- Make-Up Quiz
- More higher-order functions
- Guards
- Wrap Up

Announcements:

- HW-8 out
Exam 3 Overview

Basics:
- Closed notes, etc.
- Worth 100 points
- 6 multipart questions

General Structure: Rough mapping to questions
1. Concepts: terms, compiler/parser stages, tokens, grammars, $LL(k)$, ...
2. Parsing and AST generation
3. Semantic checking
4. VM and code generation
5. Programming paradigms and models of computation (TM’s, $\lambda$-calculus)
6. Haskell
Guards

Patterns specify “structural” conditions for matching

- Matching on the parts of a data structure
- Note: can’t check equality (or inequality) of the parts (e.g., $x:x:xs$)

Guards allow us to define “logical” conditions for a pattern

- Checking that the parts of a data structure satisfy Boolean conditions

```haskell
-- drop with just patterns (no guards)
drop _ [] = []
drop n (x:xs) = if n <= 0
    then x : xs
    else drop (n-1) xs
```

- We can rewrite this using guards to remove the if-then-else:

```haskell
  drop _ [] = []
  drop n xs | n <= 0 = xs -- if n <= 0 then xs
  drop n (_:xs) = drop (n-1) xs -- otherwise
```

- The guard gives a Boolean condition for applying the pattern
There can be *multiple* guards per pattern

```haskell
letterGrade p
    | p >= 90   = "A"
    | p >= 80   = "B"
    | p >= 70   = "C"
    | p >= 60   = "D"
    | otherwise = "F"
```

- Each guard is an expression of type `Bool`
- `otherwise` is just a binding to `True`

**How a guard works**

- For each pattern, check if first guard succeeds
- If so, RHS is result
- Otherwise, check next guard
- If no guards succeed, go to the next pattern

**When calling a function, if no patterns match ...**

- Haskell gives a runtime exception (non-exhaustive pattern)
Another (contrived) example

Q: What does this function do?

\[
\begin{align*}
\text{pairs \ [\] } &= \ [\] \\
\text{pairs \ []} &= \ [\] \\
\text{pairs \ (x:y:zs)} &= \begin{cases} \\
\quad | \ x == y &= (x,y) : \text{pairs \ (y:zs)} \\
\quad | \ \text{otherwise} &= \text{pairs \ (y:zs)} \\
\end{cases}
\end{align*}
\]

- Note: patterns and guards can be mixed (as above)
- The otherwise case here is the “default” \( x /= y \) case

Q: What is the result of \texttt{pairs \ [1,2,2,2,3]}?

\([(2,2),(2,2)]\)

Q: What is the type of \texttt{pairs}?

\[
\texttt{pairs :: (Eq a) => \[a\] \rightarrow \[(a, a)\]}
\]

Guards add another layer to checking if cases are exhaustive

Note on where with guards vs. let ... i.e., where “spans” guards for pattern

\[
\begin{align*}
\text{f \ x} &= \begin{cases} \\
\quad | \ g1 &= e1 \\
\quad | \ g2 &= e2 \\
\end{cases} \\
\text{where} \ldots
\end{align*}
\]

\[
\begin{align*}
\text{f \ x} &= \begin{cases} \\
\quad | \ g1 &= \text{let} \ldots \ \text{in} \ e1 \\
\quad | \ g2 &= \text{let} \ldots \ \text{in} \ e2 \\
\end{cases}
\end{align*}
\]
The fold function combines (accumulates) values

\( \text{foldl} \) accumulates from the left: \(((a \oplus x_1) \oplus x_2) \oplus \cdots\) 

\( \text{foldl} \) in functional form: \((f \ (f \ (f \ a \ x_1) \ x_2) \ x_3) \ x_4)\) ... for 4 elements

\( \text{foldr} \) accumulates from the right: \((x_1 \oplus (x_2 \oplus \cdots (x_n \oplus a))))\)

\( \text{foldr} \) in functional form: \((f \ x_1 \ (f \ x_2 \ (f \ x_3 \ (f \ x_4 \ a))))\) ... for 4 elements

Examples:

Prelude> foldl (+) 0 [1,2,3,4]
10

Prelude> foldl (+) 0 []
0

Prelude> foldr (-) 0 [10, 4, 3, 1]
8

Prelude> foldl (-) 0 [10, 4, 3, 1]
-18

Prelude> foldl min 10 [4, 3, 1, 5]
1

Prelude> foldl (+) 0 (map (\x -> 1) [4, 3, 1, 5])
4

Note: Here we are using a lambda function!
Defining \texttt{foldl} and \texttt{foldr} (without pattern matching)

\begin{verbatim}
foldl f a xs =
    if null xs then a
    else foldl f (f a (head xs)) (tail xs)

foldr f a xs =
    if null xs then a
    else f (head xs) (foldr f a (tail xs))
\end{verbatim}

With pattern matching:

\begin{verbatim}
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs

foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
\end{verbatim}

Q: What is the type of \texttt{foldl} (\texttt{foldr})?
Example Binary Search Tree Implementation

data Tree a = Node a (Tree a) (Tree a)  
    | Nil  
    deriving (Show, Eq)

insert :: Ord a => a -> Tree a -> Tree a
insert v Nil = Node v Nil Nil
insert v (Node x l r)
    | v <= x  = Node x (insert v l) r  
    | otherwise = Node x l (insert v r)

contains :: Ord a => a -> Tree a -> Bool
contains v Nil = False
contains v (Node x l r)
    | v == x  = True  
    | v < x  = contains v l  
    | v > x  = contains v r

erase :: Ord a => a -> Tree a -> Tree a
erase v Nil = Nil
erase v (Node x l r)
    | v < x  = Node x (erase v l) r  
    | v > x  = Node x l (erase v r)  
    | v == x  = deleteRoot (Node x l r)
where deleteRoot (Node _ Nil r) = r  
    deleteRoot (Node _ l Nil) = l  
    deleteRoot (Node _ l r) =
        let s = inorderSucc r in Node s l (erase s r)
    inorderSucc (Node x Nil _) = x  
    inorderSucc (Node _ l _) = inorderSucc l
Course Overview (From Lecture 1)

Deep dive into programming language (PL) design & implementation

- implement a “made up” typed, procedural programming language (MyPL)
- explore functional programming (using Haskell)

General course goals

- More programming experience (using ideas/techniques you've already learned)
- Better understanding of how compilers/interpreters work
- Better understanding of language design (syntax, types, constructs, trade-offs)
- Exposure to different programming “paradigms” (procedural vs functional)

Why study language implementation (“compilers”) ...

1. Essential part of computer science (and most computer-science curriculum)
2. Complicated engineering problems (example of how to build larger systems)
3. Techniques useful for a wide range of software development problems
4. Better understanding of how languages work (can improve your programming)

Why study functional programming?

1. Functional constructs have gained popularity in most (non-FP) languages
2. New ways to think about programming, new tools for problem solving