CSE 341: Programming Languages
Section AC with Nate Yazdani
recap

• regarding tail recursion, we will specifically state when you need to use tail recursion for points

• tail recursion is considered good practice in functional programming, but don’t let it bog you down otherwise

• again, if you’re unsure about your coding style, come to office hours for code review :-)

agenda

• tail recursion (review)
• anonymous and higher-order functions
• mutual recursion
• module system (if time)
standard library

- online documentation
  - http://sml-family.org/Basis/

- most useful parts
  - default stuff: http://sml-family.org/Basis/top-level-chapter.html
  - lists: http://sml-family.org/Basis/list.html
  - list pairs: http://sml-family.org/Basis/list-pair.html
  - “reals”: http://sml-family.org/Basis/real.html
  - strings: http://sml-family.org/Basis/string.html
tail recursion

- what makes a function tail-recursive?
  - its recursive calls are in tail position, i.e., tail calls

```
fun name pat = expr

if expr₁
  then expr₂
else expr₃

let val pat₁ = expr₁
  ...
  val patₙ = exprₙ
in exprₙ₊₁ end

expr handle pat₁ => expr₁

(expr₁, expr₂)
```

```
case expr₀ of
  pat₁ => expr₁
  ...
  | patₙ => exprₙ
```
tail position

a (recursive) rule of thumb for tail position:

An *subexpression* that, *if evaluated*, becomes the result of the overall expression, is in tail position.
tail-recursive fibonacci

work together to design an SML function that computes the $n^{th}$ Fibonacci number (it’s a bit tricky!)

\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2)
\end{align*}
\]
tail-recursive fibonacci

fun fib n =  
    let fun aux k =  
        if k = 1  
        then (1, 0)  
        else let val (b, c) = aux (k - 1)  
            val a = b + c  
            in (a, b) end  
    in  
        if n = 0 then 0 else #1 (aux n)  
    end
tail-recursive fibonacci

\[\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)\]

\[\text{fib}(n+1) = \text{fib}(n) + \text{fib}(n-1)\]
anonymous functions

\[
\text{fn } \text{pattern}_1 \Rightarrow \text{expression}_1 \\
| \text{pattern}_2 \Rightarrow \text{expression}_2 \\
| \ldots \\
| \text{pattern}_n \Rightarrow \text{expression}_n
\]

• an expression that evaluates to a “function value” without ever binding a name for it

• typically used to create a one-off function to pass to yet another function like \texttt{List.map}, \texttt{List.foldl}, ...

• a function that takes another function as an argument is called a \textit{higher-order function}
anonymous functions

```
fn pattern₁ => expression₁
| pattern₂ => expression₂
| ... |
| patternₙ => expressionₙ
```

- an expression that evaluates to a “function value” without ever binding a name for it
- typically used to create a one-off function to pass to yet another function like `List.map`, `List.foldl`, ...
- a function that takes another function as an argument is called a higher-order function
anonymous functions

you may hear us call these “lambda functions”

\[
\begin{align*}
| \text{pattern}_2 & \Rightarrow \text{expression}_2 \\
| \cdots \\
| \text{pattern}_n & \Rightarrow \text{expression}_n
\end{align*}
\]

• an expression that evaluates to a “function value” without ever binding a name for it

• typically you will use a one-off function to pass to yet another function like \texttt{List.map}, \texttt{List.foldl}, ...

• a function that takes another function as an argument is called a \textit{higher-order function}
currying

• two ways to create multi-argument functions
  • take a tuple for the only argument
    \[ f : t_1 \times t_2 \to t_3 \]
  • return a new function to take the next argument
    \[ f : t_1 \to t_2 \to t_3 \]

• which is better? depends on what you want
  • pro *curried*: easier to apply partially, e.g., before passing to a higher-order function
  • pro *tupled*: easier to apply altogether, e.g., for function composition
higher-order functions

please work together to do the following exercises, using anonymous functions:

1. use `map` to pair each element with itself
   ```
   map ?? [0, 1] ⊗ [(0, 0), (1, 1)]
   ```

2. use `List.filter` to get the positive integers of list
   ```
   List.filter ?? [0, 2, ~4, 3] ⊗ [2, 3]
   ```

3. use `foldl` to average an integer list
   ```
   foldl ?? ?? [2, 4] ⊗ 3
   ```
higher-order functions

please work together to do the following exercises, using anonymous functions:

1. use `map` to pair each element with itself
   ```
   map ?? [0, 1] \downarrow [(0, 0), (1, 1)]
   ```

2. use `List.filter` to get the positive integers of list
   ```
   List.filter ?? [0, 2, ~4, 3] \downarrow [2, 3]
   ```

3. use `foldl` to average an integer list
   ```
   foldl ?? ?? [2, 4] \downarrow 3
   ```
higher-order functions

please work together to do the following exercises, using anonymous functions:

1. use `map` to pair each element with itself
   \[ \text{map ?? [0, 1] \rightarrow [(0, 0), (1, 1)]} \]

2. use `List.filter` to get the positive integers of list
   \[ \text{List.filter ?? [0, 2, ~4, 3] \rightarrow [2, 3]} \]

3. use `foldl` to average an integer list
   \[ \text{foldl ?? ?? [2, 4] \rightarrow 3} \]

   kinda cheating: still need to divide afterwards
mutual recursion

• what if we need a function \( f \) to call \( g \), and a function \( g \) to call \( f \)

• this happens more often than you might think!

• a silly example, that sadly doesn’t work :-(

```
fun even x =
  x = 0 orelse not odd (x-1)
fun odd x =
  x = 1 orelse not even (x-1)
```
mutual recursion

• as clever 341 students, we may realize that higher-order functions offer a work-around

```java
fun even (odd, x) =
    x = 0 orelse not odd (even, x-1)
fun odd (even, x) =
    x = 1 orelse not even (odd, x-1)
```

• this doesn’t feel like a great solution, though
mutual recursion

• as clever 341 students, we may realize that higher-order functions offer a work-around

  each function passes itself to the other

  \[
  \begin{align*}
  \text{fun even} & \ (\text{odd}, \ x) = \\
  & \ x = 0 \ \text{orelse} \ \text{not} \ \text{odd} \ (\text{even}, \ x-1) \\
  \text{fun odd} & \ (\text{even}, \ x) = \\
  & \ x = 1 \ \text{orelse} \ \text{not} \ \text{even} \ (\text{odd}, \ x-1)
  \end{align*}
  \]

• this doesn’t feel like a great solution, though
mutual recursion

• SML has a special keyword to help us out

```ml
fun even x =
  x = 0 orelse not odd (x-1)
and odd x =
  x = 1 orelse not even (x-1)
```

• also works with mutually recursive `datatype` bindings

```ml
datatype even = Zero | ESucc of odd
and odd = OSucc of even
```
mutual recursion

• SML has a special keyword to help us out

```ml
fun even x = 
  x = 0 orelse not odd (x-1)
and odd x = 
  x = 1 orelse not even (x-1)
```

• also works with mutually recursive `datatype` bindings

```ml
datatype even = Zero | ESucc of odd
and odd = OSucc of even
```

I fully admit that this is a contrived example :-(