CS 421 Midterm review session

- Outline
  - Overview
  - Your questions
    - General
    - Sample MT problems

Overview

- Format
  - 75 minutes
  - Closed-book, closed-notes
  - No calculators, no phones, no computers, no talking
  - No clarifications
- Content:
  - MPs
  - Lecture examples
  - Lecture slides
  - Mostly analysis + synthesis, not recall

OCaml: tail recursion

- No further computation follows the recursive call
- TR example (MP2 p4):

```
let rec concat_even l =
  match l with
    []     -> ""
  | s::[] -> ""
  | s::s'::ss -> s' ^ concat_even ss ;;
```

- Non-TR example (Fibonacci):

```
let rec fib n =
  match n with
    0  -> 1
  | 1  -> 1
  | _  -> fib (n-1) + fib (n-2)
```

OCaml: nested let

```
let f x y =
      let z = sqrt(x+y)
      in x*z;;
```

```
let f x y =
    let z = …
    and t = …
    in … z … t …
```

OCaml: currying

```
let f x y = x + y
```

```
let f (x,y) = x + y
```

OCaml: list folding

- Not on the midterm
  - Higher-order functions

OCaml: function types

- What is the datatype of (and how do we know):

```
let f g = match g with
   (x::y)::z -> y
 | a::b -> b

This function is wrong!
f: 'a list list -> ('a list) or ('a list list)


let rec h a b =
       if a = [b]
       then true
       else h a (tl b)

h: 'a list list -> 'a list -> bool
```

Grammars: EBNF => EBNF

- Only differences (that we care about):
  - *
  - +
  - ?

- Example:

```
A -> B* | C+ | D?

=>

A -> A1 | A2 | A3
A1 -> B A1 | “”
A2 -> C A2 | C
A3 -> D | “”
```

Top-down parsing: LL(1) condition

- "Can do recursive descent by looking only at the next lookahead token"
  - No left recursion
  - Pairwise distinct FIRST sets
- FIRST (X)
  - X – non-terminal
  - FIRST(X) – possible starting terminal, or "" if *nullable*
- Example

```
F -> id ( A )
A -> "" | B
B -> id id | B, id id

FIRST(F) = {id}, FIRST (A) = {id, ""}, FIRST (B) = {id}
```

Top-down parsing: Some/None

- parseA : token list -> (token list) option
- type ‘a option = None | Some ‘a

Top-down parsing: associativity

- Is there any easy way of guessing / figuring out whether an expression is left- or right- associative?
  - No
  - Have to know the meaning (semantics) of the language
  - We typically use canonical math expressions, or Java

- Can tell whether a *grammar* is left- or right-associative
  - A -> id + A | id
  - A -> A + id | id

Ocaml: sample mt 2c

- Implement the Ocaml function partition: int list -> (int list) list, which divides a list into “runs” of the same integer, e.g.
  - partition [9;9;5;6;6;6;3] = [[9;9]; [5]; [6;6;6]; [3]]
- Solution

```
let rec partition lis =
      if lis = [] then []
      else match partition (tl lis) with
           [] -> [[hd lis]]
         | x :: xs -> if hd lis = hd x
                      then (hd lis :: x) :: xs
                      else [hd lis] :: (x :: xs)
```

Ocaml: sample mt 2f

- compress: int list -> (int * int) list replaces runs of the same integer with a pair giving the count and the number. E.g.
  - compress [1;1;5;6;6;6;3] = [(2,1); (1,5); (3,6); (1,3)]
- Solution

```
let rec compress lis = if lis = [] then [] else
     match compress (tl lis) with
       [] -> [(1, hd lis)]
     | (n,x)::lis' -> if x = hd lis
                  then (n+1,x):: lis'
                  else (1, hd lis)::(n,x)::lis'
```

Ocaml: sample mt 3ii

- type btree = Leaf of int | Node of int * btree * btree
- followpath: btree -> boolean list -> int list gives the list of integers in the tree on the path described by the boolean list, where "true" means follow the left child and "false" means follow the right child.

- Solution

```
let rec followpath bt blis = match bt with
      Leaf n -> [n]
    | Node(n,lt,rt) -> if blis = [] then [n]
                    else if hd blis
                       then n::(followpath lt (tl blis))
                       else n::(followpath rt (tl blis))
```

Top-down parsing: sample mt 5

- Grammar:
  - E -> E + T | T
  - T -> T * P | P
  - P -> id | (E)
- Definitions:
  - type tree = Node of string * tree list
  - type exp = Id of string | Plus of Exp*Exp | Times of Exp*Exp

Top-down parsing: sample mt 5

- Solution

```
let rec abstract t = match t with
  Node(s, children) ->
      (match children with
       [] -> Id s
     | [ch] -> abstract ch
     | [Node("(",[]); ch; Node(")",[])] -> abstract ch
     | [ch1; Node(op, []); ch2] ->
               let ach1 = abstract ch1
               and ach2 = abstract ch2
               in if op = "+" then Plus(ach1, ach2)
                          else Times(ach1, ach2) )
```

Bottom-up parsing: sample mt 13b

- Using precedence to disambiguate grammars
  - E -> E . id | ! E | id
- Ambiguity: ! a . b
- Disambiguate:
  - %nonassoc !
  - %right .

Top-down parsing: sample mt 15d

- Translate grammar to LL(1)

```
F -> id ( A )
A -> ε | B
B -> id id | B, id id
```

  - Problem: left-recursive.
- LL(1) version

```
F -> id ( A )
A -> ε | id id B
B -> ε | , id id B
```

- Implement parser
  - Purely mechanical transformation, based on the above rules

```
let rec parseF lis = match lis with
        Id::LParen::lis’ -> (match parseA lis’ with
                               Some (RParen :: lis’’) -> Some lis’’
                             | _ -> None)
      | _ -> None
```

OCaml: su08 mt 2d

- Function pairs:

```
let rec pairs a bs =
  match bs with
    | [] -> []
    | x::xs -> (a,x) :: pairs a xs
```

- What is the result (English-language description) of pairs 3?
  - ’a list -> (int * ’a) list = <fun>
  - A function that, given a list of items, will return a new list made up of those items paired with 3 (pairs of the form (3, b), where b is an item from the list).

Ocaml: su08 mt 2e, 3

- Not on midterm
  - Higher-order functions

Grammars: su08 mt 6a

- Grammar:

```
E -> E + T          T -> T * F          F -> id
E -> E - T          T -> T/F            F -> num
E -> T              T -> F
```

- Show a leftmost derivation for the following term:

```
x * y + (5 - 3)

E => E + T => T + T => T * F + T => F * F + T => x * F + T => x * y + T
=> x * y + F => x * y + (E) => x * y + (E - T) => x * y + (T - T)
=> x * y + (F - T) => x * y + (5 - T) => x * y + (5 - F)
=> x * y + (5 - 3)
```

Types: su08 mt 7

- Not on the midterm
  - Type derivations