CMSC 631 Program Analysis and Understanding

Bill Pugh

Administrivia

- Designed as a core graduate course
  - prepares you for more specific, project oriented graduate courses in programming languages
  - new course, but expect to offer it once a year
  - grades based on exams and homeworks
    - some homeworks will include programming projects

Course material

- Designed to prepare you for research in
  - compilers and code generation
  - exploring new programming languages
  - software tools to detect errors or aid in program understanding

Plchat Seminar

- This semester, Mondays, 11am, CSIC 3120
- Mailing list
  - including other PL-related announcements
- First talk next Monday:
  - Cyclone: A next-generation systems programming language, by Mike Hicks, Univ. of Maryland

Quiz: show of hands

- How many have programmed in:
  - some version of ML (SML or O’Caml)?
  - Haskell or another lazy functional language?
  - Smalltalk or Squeak?
  - OO-language other than C++, Java or C#?
  - Used Generic Polymorphic types in Java?

Quiz: show of hands

- Have been exposed to:
  - contravariant types
  - data flow analysis (e.g., def-use chains)
  - abstract syntax trees
  - register allocation
  - design patterns
Course material

- Tentative topics on web page
- Initial topics:
  - Types
  - OCaml

O’Caml

- Objective Caml is a fast modern type-inferring functional programming language descended from the ML (Meta Language) family. The OCaml compiler was developed at INRIA’s projet Cristal.

Learning OCaml

- After some intro stuff today, you will be largely responsible for learning OCaml on your own
- Installed on /fs/imports on CS Suns
- May get cluster account for class if needed
  - not a good environment

ocaml

- Lots of OCaml tools (e.g. compiler)
- ocaml is the interactive interpreter
- Enter expressions, terminated with a ;;
  - I don’t know why ;; is used as a terminator

Operators and Functions

- Integer math
  - +, -, *, /, mod
- Real math
  - +, -, *, /, **
- Logical
  - <, <=, >, >=, =
- Bitwise
  - land, lor, lxor, lsl, lsr, asl
  - if then else

Conversion functions

- float, int_of_float
- int_of_char, char_of_int
- string_of_int, int_of_string

Definitions and Functions

- let pi = 4.0 *. atan 1.0;;
- let x = 5 in x*x;;
- let plus x y = x+y;;
- let incr x = plus 1 x;;
- Recursive definition:
  - let rec fib n = if n < 2 then 1 else fib(n-1) + fib(n-2);;
Lists

• \([1; 2; 3; 4]\)
• Empty list is \([\,]\)
• Use :: to add to front
  – \(1 :: [2 ; 3 ; 4] = [1; 2; 3; 4]\)

match

• O’Caml equivalent of case statement
• Often switch on type structure
• let rec insert elt lst = match lst with
  \([\,] \rightarrow \text{elt}\)
  | head :: tail -> if elt <= head then elt :: lst
  else head :: insert elt tail;;

More on matching

• let rec fib i = match i with
  0 \rightarrow 1
  | 1 \rightarrow 1
  | i \rightarrow fib(n-1) + fib(n-2);;
• Also:
  – let fib = function
    0 \rightarrow 1
    | 1 \rightarrow 1
    | i \rightarrow fib(n-1) + fib(n-2);;

currying

• If you have a function that takes \(n\) arguments
• providing just one argument to it returns a function that takes \(n-1\) arguments

Anonymous functions

• fun x -> x*x;;
• same as [\(\lambda x. x*x\)]

Higher order functions

• let compose f g x = f ( g ( x ) );;
• let twice f = compose f f;;
• let rec map f l = match l with
  \([\,] \rightarrow [\,]
  | hd :: tail -> f hd :: map f tl;;
• Type variables denoted ‘\(a\), ‘\(b\), …
Variant types

- type 'a btree = Empty
  | Node of 'a * 'a tree * 'a tree;;
- matching variants:
  - let rec member x btree = match btree with
    Empty -> false
    | Node (y, left, right) -> if x = y then true else if x < y then member x left else member x right ;;

Matching

- If multiple cases match, only the first applies
- Pattern match variables must be unique
  – can’t match [ x; x ]
- Can use _ for don’t care patterns

Alternative member

- let member x btree = let rec find = function
  Empty -> false
  | Node (y, left, right) -> if x = y then true else if x < y then find left else find right
  in find btree ;;
- using function rather than match
- uses closures: find references variable x from outer scope
  – this use doesn’t require creating closures
  – others do, e.g., let add x = fun y -> x+y ;;

insert into binary trees

- let rec insert x = function
  Empty -> Node(x, Empty, Empty)
  | Node(y, left, right) -> if (x < y) then Node(y, insert x left, right) else Node(y, left, insert x right);;

incomplete matches

- let is_uppercase = function
  'A' .. 'Z' -> true
  | 'a' .. 'z' -> false ;;
- Complains at compile time
  – throws an exception at runtime

Strings

- Use double quoted string constants
- Use ^ for concatenation
- use .[i] for the i’th character (same indexing syntax as arrays).
Printing, access to command line

```ml
let args = Sys.argv in
if Array.length args = 2 then
  let n = int_of_string Sys.argv.(1) in
  print_int (n * n);
  print_newline ();
else
  print_string "Usage: square <number>n";;
```

#use

- Like an include file
- `#use "defs.ml";;`
- only at top level
- Other mechanisms for building large systems
  - won’t be covered today

compiling, debugging

- `ocamlc -o test test.ml`
  - compiles test.ml to an executable
- `-g` flag adds debugging information
- `ocamldebug` provides debugging