

# Lecture 18 — Implementation of MiniOCaml

- MiniOCaml — Subset of OCaml
- Description of MiniOCaml
- MP8: Two interpreters for dynamically-typed version of MiniOCaml
  - Substitution model
  - Environment-based model

# Functional programming

- Two essential features of functional programming:
  - Recursion over lists and trees
  - Higher-order functions, a.k.a. first-class functions
    - Treat functions the same as other values: pass as arguments, put into lists and tuples, etc.
  - *We will concentrate on this for the next two weeks.*
- Why study f.p.?
  - Different perspective on programming
  - Increasingly used in practice (e.g. in scripting languages)
  - Shows up in some of the trickiest parts of regular languages, e.g. C++ STL

# MiniOCaml

- Like OCaml, but without: pattern-matching; mutual recursion (“and”); type definitions (only built-in types — ints, floats, bools, strings, pairs, lists)
- Concrete syntax of MiniOCaml (start symbol is *exp*):

$$\begin{aligned} \text{exp} \rightarrow & \text{exp binary-operation exp} \mid \text{unary-operation exp} \\ & \mid \text{INTEGER\_LITERAL} \mid \text{FLOAT\_LITERAL} \mid \text{STRING\_LITERAL} \mid \text{TRUE} \mid \text{FALSE} \\ & \mid \text{LBRACK explis1 RBRACK} \mid \text{LPAREN explis2 RPAREN} \mid \text{LET def IN exp} \\ & \mid \text{IDENT} \mid \text{IF exp THEN exp ELSE exp} \mid \text{LET REC def IN exp} \mid \text{exp exp} \\ & \mid \text{FUN IDENT RIGHTARROW exp} \end{aligned}$$

$$\text{def} \rightarrow \text{IDENT args EQ exp}$$

$$\text{args} \rightarrow \epsilon \mid \text{args IDENT}$$

$$\text{binary-operation} \rightarrow \text{COMMA} \mid \text{EQ} \mid \text{LT} \mid \text{GT} \mid \text{NEQ} \mid \text{ANDAND} \mid \text{etc.}$$

$$\text{unary-operation} \rightarrow \text{NOT} \mid \text{HD} \mid \text{TL} \mid \text{FST} \mid \text{SND} \mid \text{etc.}$$

$$\text{explis1} \rightarrow \text{list of exp's separated by semicolons}$$

$$\text{explis2} \rightarrow \text{list of exp's separated by commas}$$

# Anonymous functions

- One new syntactic form we haven't seen before (but it is in OCaml): “`fun x -> exp`”
- Denotes a function that takes `x` to the value of `exp` (which presumably contains `x`).

```
(fun a -> a*a) 4
```

```
let addnl = fun s -> s ^ "\n" in addnl "abc"
```

```
let funlist = [fun a -> a+1; fun a -> a*a]  
in (hd (tl funlis)) 3
```

```
let funpair = (fun x -> x+1, fun y -> y+3)  
in (fst funpair) 4 + (snd funpair) 5
```

```
let rec fac = fun x -> if x=0 then 1 else x * fac (x-1) in fac 5
```

# Exercises with anonymous functions

- Define a list of three functions: the first adds 1 to any integer; the second doubles any integer; and the third triples any integer. Then apply the second function to 4:

```
let funlis =  
  [      ;      ;      ]  
  
in (      ) 4
```

- Transform the following expression to the form `let rec length = ...`, and remove pattern-matching:

```
let rec length lis = match lis with [] -> 0 | h::t -> 1 + length t  
in length [1;2;3]
```

# Multi-argument functions

- MiniOCaml, like OCaml, has multi-argument functions. But it is important to realize that they *are not necessary*:

- Transform

`let f x = e1 in e2`

to:

`let f = fun x e1 in e2`

- The abstract syntax of MiniOCaml, which we will look at shortly, has only single-argument functions.

# MP 8 — Interpreters for dynamically-typed MiniOCaml

- We will provide lexer/parser and translation to abstract syntax
- You will write two functions to evaluate expressions:
  - `reduce`: map *closed* expressions to values by simplification
  - `eval`: map expressions *and* environments to values (as in MP5)
- As in MP5, we will provide precise evaluation rules for both versions
- Today we are discussing `reduce`, a.k.a. the “substitution model”

# Abstract syntax of MiniOCaml

```
type exp =  
  Operation of exp * binary_operation * exp  
| UnaryOperation of unary_operation * exp  
| Var of string | StrConst of string | IntConst of int  
| FloatConst of float | True | False  
| List of exp list | Tuple of exp list  
| If of exp * exp * exp | App of exp * exp  
| Let of string * exp * exp  
| Fun of string * exp  
| Rec of string * exp  
  
and binary_operation = Semicolon | Comma | Equals | LessThan  
| GreaterThan | NotEquals | Assign | And | Or  
| IntPlus | IntMinus | IntDiv | IntMult  
| FloatPlus | FloatMinus | FloatDiv | FloatMult  
| StringAppend | ListAppend  
  
and unary_operation = Not | Head | Tail | Fst | Snd
```

# Notes on the abstract syntax

- `let f x = e1 in e2` **becomes** `Let("f", Fun("x", e1), e2)`
- `let f x y = e1 in e2` **becomes** `Let("f", Fun("x", Fun("y", e1)), e2)`
- `let rec f x = e1 in e2` **becomes** `Let("f", Rec("f", Fun("x", e1)), e2)`. **(The second argument of Rec will *always* be a Fun.)**
- `Tuple` allows a tuple of any length; however, there is no **pattern-matching** to define operations on tuples, and operations `fst` and `snd` apply only to pairs, so only pairs are **usable**.

# Evaluation by substitution

- One of the virtues of functional languages is that their execution can be understood as simplification — transforming expressions (ASTs) to simpler expressions (ASTs).
- The main idea is to recursively simplify arguments until a simple value is obtained, and then apply an operation on that value:

$$\begin{array}{c} e_1 \text{ op } e_2 \Downarrow v_1 \text{ OP } v_2 \\ e_1 \Downarrow v_1 \\ e_2 \Downarrow v_2 \end{array}$$

- Function application and recursion are a little trickier.
- Note that we can only simplify *closed* expressions — those with no free variables.

# Exercise: define closed

```
(* closed e is true iff e has no free variables.  
   closed1 e varlist is true iff any free variables in e are in varlist  
   (Assume function member: string -> string list -> bool)   *)
```

```
let closed (e:exp) : bool =
```

```
  let rec closed1 (e:exp) (vars:string list) : bool =  
    match e with
```

```
  in closed1 e []
```

# Rules for evaluation by substitution

$\text{IntConst } i \Downarrow$

$e_1 + e_n \Downarrow$

$\text{List } [e_1, \dots, e_n] \Downarrow$

# Values

- Before continuing, need to be more precise about what we mean by a “value” — how do we know when we’re done simplifying an expression?
- Values are ASTs that are considered to be so simple that they cannot be simplified further. They are:
  - All constants — `IntConst  $i$` , `FloatConst  $x$` , `StrConst  $s$` , `True`, `False`
  - Lists or tuples: `List  $[v_1, \dots, v_n]$`  or `Tuple  $[v_1, \dots, v_n]$` , so long as  $v_1, \dots, v_n$  are values.
  - `Fun(x,e)` is always a value.
- If  $e \Downarrow e'$ , then  $e'$  is a value. Similarly, `reduce:exp  $\rightarrow$  exp` always returns a value.

# Values (cont.)

- Non-values are AST terms that should be further simplified: any terms with operators `If`, `Let`, `App`, `Operation`, or `UnaryOperation`, as well as `List` and `Tuple` (if their arguments are not values).
- (`Rec` is a bit in-between: it requires exactly one step of simplification, as we will see.)
- Note that previous rules for `IntConst`, `List`, and `Tuple` always produced values.
- What about `Var  $x$` ?
- Why this is called the “substitution model”: apply function by substituting value of argument into body of function. Use notation  $e[v/x]$  for replacing all  $x$ ’s in  $e$  with  $v$ .

# Rules for evaluation by substitution (cont.)

$\text{If}(e_1, e_2, e_3) \Downarrow$

$\text{Fun}(x, e) \Downarrow$

# Rules for evaluation by substitution (cont.)

$\text{Let}(x, e_1, e_2) \Downarrow$

$e_1 \ e_2 \Downarrow$

# Examples of evaluation by substitution

```
let x = 3 in x+1
```

```
(fun x -> x+1) 3
```

# Examples of evaluation by substitution

```
((fun x -> fun y -> x+y) 3) 4
```

```
let f = fun x -> fun y -> x+y  
in let g = f 1  
   in g 2
```

# Rules for evaluation by substitution (cont.)

- Definition of substitution needs to be given precisely — see MP8. However, its meaning is intuitively clear:  $e[v/x]$  means to replace all occurrences of  $x$  in  $e$  by  $v$ . Just remember that if  $e$  contains a subexpression that binds  $x$  (a Let, Fun, or Rec), then leave that subexpression alone.
- One last rule:

$$\text{Rec}(f, \text{Fun}(x, e)) \Downarrow \text{Fun}(x, e[\text{Rec}(f, \text{Fun}(x, e))/f])$$

# Examples of evaluation by substitution

```
let map = fun f -> Rec(map1, fun lis -> if lis=[] then []  
                                     else (f (hd lis)) :: map1 (tl lis))  
in map (fun x->x+1) [1;2;3]
```

# map and fold\_right

- **map and fold\_right (often called reduce) are the best known *higher-order functions*.**
- **Each takes a function as an argument, and applies it to the elements of a list.**
- $\text{map } f [x_1; \dots; x_n] = [f \ x_1; \dots; f x_n]$
- $\text{fold\_right } f [x_1; \dots; x_n] a = f \ x_1 (f \ x_2 (\dots (f \ x_{n-1} (f \ x_n a)) \dots ))$ .