CS251 Programming Languages **Handout # 37**
Prof. Lyn Turbak **April 13, 2007**
Wellesley College

FOFL and FOBS: First-Order Functions

We have seen the power of first-class higher-order functions in OCAML, and have seen how they can be implemented in the mini-language HOFL. The functions (likewise procedures, methods, and subroutines) in most real-world programming languages (including C, JAVA, BASIC, PASCAL, and FORTRAN) are much more limited than those in OCAML and HOFL. In this handout, we explore two mini-languages with more limited kinds of functions:

1. FOFL (First-Order Functional Language) extends VALEX with first-order second-class global functions. Functions in FOFL are similar to those in C[1].
2. FOBS (First-Order Block-Structure Language) extends FOFL with block structure – the ability to declare functions inside of other functions. Functions in FOBS are similar to those in PASCAL[2].

In both of these mini-languages, functions are **first-order** — they are second-class entities that cannot be passed as arguments to functions, returned as results from functions, or stored in data structures.

1 FOFL

1.1 Syntax of FOFL

FOFL extends VALEX with globally-defined functions. The full grammar of FOFL is presented in Fig. 1. The key new features that FOFL adds to VALEX are:

- *Global Function Declarations* In addition to the program parameters and body expression, FOFL programs include an arbitrary number of mutually recursive global function declarations of the form `(def (`F_{name} I_{formal_1} ... I_{formal_n}`)` E_{body}`)`. Here, F is a meta-variable that ranges over function names. These are a different class of names than the variable names ranged over by the identifier meta-variable I. In FOFL, functions may be declared only in the top-level `fofl` program construct.

  The function declaration syntax has been chosen so that FOFL programs have the form of restricted HOFL programs. Whereas top-level `def`s desugar into `bindrec` in HOFL, they are kernel forms in FOFL.

- *Function Applications* A globally declared function can be applied in a function application expression (funapp) that has the form `(`F_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`. Here, F_{rator} is the *name* of the function being applied and $E_{rand_1} \ldots E_{rand_n}$ are the expressions denoting its arguments.

Here are the standard factorial and Fibonacci functions expressed in FOFL:

```
(fofl (x) (fact x)
  (def (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
  )
```

[1] C supports a limited form of first-class functions in the form of function pointers; this important feature is *not* modeled in FOFL.

[2] PASCAL allows functions to be passed as arguments but not returned as results; this important feature is *not* modeled in FOBS.

$P \in$ *Program*

$P \rightarrow$ `(fofl (`I_{formal_1} `...` I_{formal_n}`)` E_{body} $FD_1 \ldots FD_k$`)`	Program

$FD \in$ *Function Declaration*

$FD \rightarrow$ `(def (`F_{name} I_{formal_1} `...` I_{formal_n}`)` E_{body}`)`	Function Declaration

$E \in$ *Expression*

Kernel Expressions:

$E \rightarrow L$	Literal
$E \rightarrow I$	Variable Reference
$E \rightarrow$ `(if` E_{test} E_{then} E_{else}`)`	Conditional
$E \rightarrow$ `(bind` I_{name} E_{defn} E_{body}`)`	Local Binding
$E \rightarrow$ `(`O_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`	Primitive Application
$E \rightarrow$ `(`F_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`	Function Application

Sugar Expressions:

$E \rightarrow$ `(&&` E_1 E_2`)`	Short-Circuit And
$E \rightarrow$ `(||` E_1 E_2`)`	Short-Circuit Or
$E \rightarrow$ `(cond (`E_{test_1} E_{body_1}`) ...` `(`E_{test_n} E_{body_n}`) (else` $E_{default}$`))`	Multi-branch Conditional
$E \rightarrow$ `(bindseq ((`I_{name_1} E_{defn_1}`) ...` `(`I_{name_n} E_{defn_n}`))` E_{body}`)`	Sequential Binding
$E \rightarrow$ `(bindpar ((`I_{name_1} E_{defn_1}`) ...` `(`I_{name_n} E_{defn_n}`))` E_{body}`)`	Parallel Binding
$E \rightarrow$ `(list` E_1 `...` E_n`)`	List
$E \rightarrow$ `(quote` S`)`	Quoted Expression

$S \in$ *S-expression*

$S \rightarrow N$	S-expression Integer
$S \rightarrow C$	S-expression Character
$S \rightarrow R$	S-expression String
$S \rightarrow I$	S-expression Symbol
$S \rightarrow$ `(`S_1 `...` S_n`)`	S-expression List

$L \in$ *Literal*

$L \rightarrow N$	Numeric Literal
$L \rightarrow B$	Boolean Literal
$L \rightarrow C$	Character Literal
$L \rightarrow R$	String Literal
$L \rightarrow$ `(sym` I`)`	Symbolic Literal
$L \rightarrow$ `#e`	Empty List Literal

$O \in$ *Primitive Operator*: e.g., `+`, `<=`, `and`, `not`, `prep`
$F \in$ *Function Name*: e.g., `f`, `sqr`, `+-and-*`
$I \in$ *Identifier*: e.g., `a`, `captain`, `fib_n-2`
$N \in$ *Integer*: e.g., `3`, `-17`
$B \in$ *Boolean*: `#t` and `#f`
$C \in$ *Character*: `'a'`, `'B'`, `'7'`, `'\n'`, `''''\\'`
$R \in$ *String*: `"foo"`, `"Hello there!"`, `"The string \"bar\""`

Figure 1: Grammar for the FOFL langauge.

```
(fofl (x) (fib x)
  (def (fib n)
    (if (<= n 1)
        n
        (+ (fib (- n 1))
           (fib (- n 2)))))
  )
```

The globally declared functions in a FOFL program are mutually recursive:

```
(fofl (n) (list (even? n) (odd? n))
  (def (even? x)
    (if (= x 0)
        #t
        (odd? (- x 1))))
  (def (odd? y)
    (if (= y 0)
        #f
        (even? (- y 1))))
  )
```

Here is a simple example of list processing in FOFL:

```
(fofl (a b) (sum (map-square (filter-even (range a b))))
  (def (sum ns)
    (if (empty? ns)
        0
        (+ (head ns)
           (sum (tail ns)))))
  (def (map-square ns)
    (if (empty? ns)
        #e
        (prep (* (head ns) (head ns))
              (map-square (tail ns)))))
  (def (filter-even ns)
    (if (empty? ns)
        #e
        (if (even? (head ns))
            (prep (head ns) (filter-even (tail ns)))
            (filter-even (tail ns)))))
  (def (range lo hi)
    (if (> lo hi)
        #e
        (prep lo (range (+ lo 1)) hi)))
  (def (even? n) (= 0 (% n 2)))
  )
```

Functions in FOFL are **second-class**: Although they can be named by function declarations, they *cannot* be passed as arguments to other functions, returned as values from functions, stored in lists, or created anywhere other than in a top-level function declaration. We call such second-class functions **first-order functions**.

1.2 Namespaces

A programming language may have several different categories of names. Each such category is called a namespace. For example, JAVA has distinct namespaces for packages, classes, methods, instance variables, class variables, and method parameters/local variables.

In a language with multiple namespaces, the same name can simultaneously be used in different namespaces without any kind of naming conflict. For example, consider the following JAVA class declaration:

```
public class Circle {

  // Instance variable of a Circle object.
  public double radius;

  // Constructor method for creating Circle objects.
  public Circle (double r) {
    this.radius  = r;
  }

  // Instance  method for scaling Circles.
  public Circle scale (double factor) {
    return new Circle(factor * this.radius);
  }
}
```

It turns out that we can rename every one of the names appearing in the above program to `radius` (as shown below) and the class will have the same meaning!

```
public class radius {

  // Instance variable of a circle object.
  public double radius;

  // Constructor method for creating Circle objects.
  public radius (double  radius) {
    this.radius  =  radius;
 }

  // Instance  method for scaling Circles.
  public radius radius (double radius) {
    return new radius(radius * this.radius);
  }
}
```

Of course, in order to use the renamed class, we would need to change uses of the original class consistently. For instance, the expression `(new Circle(10)).scale(2).radius` would have to be renamed to `(new radius(10)).radius(2).radius`.

Although using the name `radius` to stand for entities in four different namespaces (class, instance variable, instance variable name, parameter name) would make the program very difficult for a human program to read, the JAVA compiler and JAVA bytecode interpreter treat the renamed program identically to the original.

JAVA has an unusually high number of namespaces. But many languages have at least two namespaces: one for functions, and one for variables. For instance, in this category are PASCAL and COMMON LISP, as well as the mini-languages FOFL and FOBS that we are studying. In contrast, many functional languages, such as SCHEME, ML, and HASKELL (as well as the mini-language HOFL) have a single namespace for functions and variables. This is parsimonious with the first-classness of functions, which allows functions to be named like any other values.

As a somewhat silly example, consider the following working definition of a factorial function in FOFL:

```
(def (fact fact)
  (if (= fact 0)
      1
      (* fact (fact (- fact 1)))))
```

In this example, there are two distinct entities named `fact`: the factorial function (in the function namespace) and the formal parameter of the factorial function (in the variable namespace). Because the namespaces are distinct, there is no confusion between the entities. If the same experiment were tried in HOFL, OCAML, SCHEME, or C, however, the function would encounter an error when applied to a number because all occurrences of `fact` in the body — including the one in the operator position — would refer to a number.

1.3 Evaluation and Scope in FOFL

A complete environment-model evaluator for FOFL is presented in Fig. 2. Highlights of the evaluator include:

- There are three environments:

  1. The **function environment** `fenv` represents the function namespace. Function names are looked up here in the `apply` function.
  2. The **value environment** `venv` represents the value namespace that binds program parameters, function parameters, and `bind` names. Variable names are looked up here in the `Var` clause of `eval`.
  3. The **global environment** `genv` is that portion of the value environment that binds only the program parameters.

- Just as in HOFL, FOFL has scoping mechanisms that determine the meaning of a free variable in a function. `env-run` is parameterized over a `scope` parameter that determines the parent environment of the application frame created by a function application. The parent environment is determined from `genv` and `venv`. This supports the two traditional forms of scoping:

  – `static`: The parent environment is `genv`.
  – `dynamic`: The parent environment is `venv`.

  It also allows for some non-traditional forms of scoping:

  – `empty`: The parent environment is the empty environment.
  – `merged`: The parent environment is the result of merging `genv` and `venv`.

  Note that another option is to make the parent environment the result of merging `venv` and `genv`; but this is equivalent to dynamic scope.

As examples of FOFL scoping, consider running the following two programs on the argument list `[3]` under the four different scoping mechanisms:

```
; Program 1
(fofl (a) (try 100)
  (def (add-a x) (+ x a))
  (def (try a) (add-a (* 2 a)))
  )
```

```
(* Model a FOFL scoping mechanism as a way of combining global and local environments *)
type scoping = valu Env.env (* global parameter environment *)
               * valu Env.env (* local parameter environment *)
               -> valu Env.env

(* val run : scoping -> Fofl.pgm -> int list -> valu *)
(* Note: this function is a compelling example of block structure in OCAML! *)
let rec run scope (Pgm(fmls,body,fcns)) ints =
  let flen = length fmls
  and ilen = length ints
  in if flen <> ilen then
       raise (EvalError ("Program expected " ^ (string_of_int flen)
                         ^ " arguments but got " ^ (string_of_int ilen)))
     else
       let genv = Env.make fmls (map (fun i -> Int i) ints) (* global param env *)
       and fenv = Env.make (map fcnName fcns) fcns (* function env *)
       in let rec eval exp venv (* current variable env *) =
            match exp with
              Lit v -> v
            | Var name ->
                (match Env.lookup name venv with
                  Some(i) -> i
                | None -> raise (EvalError("Unbound variable: " ^ name)))
            | PrimApp(op, rands) -> (primopFunction op) (map (flip eval venv) rands)
            | If(tst, thn, els) ->
                (match eval tst venv with
                  Bool b -> if b then eval thn venv else eval els venv
                | v -> raise (EvalError ("Non-boolean test value " ^ (valuToString v)
                                         ^ " in if expression")))
            | Bind(name, defn, body) -> eval body (Env.bind name (eval defn venv) venv)
            | App(fname, rands) -> apply fname (map (flip eval venv) rands) venv

            and apply fname actuals venv =
              match Env.lookup fname fenv with
                None -> raise (EvalError ("unknown function " ^ fname))
              | Some (Fcn(name,formals,body)) ->
                  let flen = length formals and alen = length actuals
                  in if flen <> alen then
                    raise (EvalError ("Function " ^ name ^ " expected "
                                      ^ (string_of_int flen) ^ " arguments but got "
                                      ^ (string_of_int alen)))
                  else eval body (Env.bindAll formals actuals (scope genv venv))
          in eval body genv (* program body is evaluated in global environment *)

(* Scoping mechanisms *)
let static = fun genv venv -> genv
let dynamic = fun genv venv -> venv
(* "weird" scopes *)
let empty = fun genv venv -> Env.empty
let merged = fun genv venv -> Env.merge genv venv
(* Note that Env.merge venv genv is equivalent to dynamic scope *)
```

Figure 2: An environment model evaluator for FOFL parameterized over the scoping mechanism (`scope`) for free value variables in a function.

```
; Program 2
(fofl (a) (test (* 100 a) (* 10 a))
  (def (linear x) (+ (* a x) b))
  (def (test a b) (linear 2))
  )
```

Scope	Value in Program 1	Value in Program 2
`static`		
`dynamic`		
`empty`		
`merged`		

2 FOBS

FOBS extends FOFL with **block structure** – the ability to locally declare within functions any kind of declaration that can be made at top-level. In particular, since FOFL supports top-level function declarations, FOBS allows functions to be declared within functions. Here we first explore block structure in the context of HOFL, and then discuss block structure in FOFL.

2.1 Block Structure in HOFL

Block structure in HOFL is realized via two constructs:

- HOFL's `abs` construct allows creating functions anywhere in a program, even within other functions.
- HOFL's `bindrec` construct allows creating collections of mutually recursive functions anywhere in a program, even within other functions.

Recall that HOFL's `fun` construct desugars into nested instances of `abs` and that top-level HOFL declarations of the form

`(def (`I_{name} I_{formal_1} ... I_{formal_n}`)` E_{body}`)`

desugar into a `bindrec` of `fun`s.

As a simple example of block structure, consider the following HOFL function declaration:

```
(def (index-bs x xs)
  (bindrec ((loop (fun (i ys)
                    (if (empty? ys)
                        -1
                        (if (= x (head ys))
                            i
                            (loop (+ i 1) (tail ys)))))))
    (loop 1 xs)))
```

Note how the local `loop` function can refer to the parameter `x` of the enclosing function declaration even though it is not passed as an explicit parameter.

The above program can be expressed without block structure by passing `x` as an explicit parameter to the `index-loop` function:

```
(def (index-no-bs x xs)
  (index-loop 1 x xs))

(def (index-loop i ys x)
  (if (empty? ys)
      -1
      (if (= x (head ys))
          i
          (index-loop (+ i 1) (tail ys) x))))
```

As another example of block structure, consider a block-structured version of a function calculating cartesian products:

```
(def (cartesian-product-bs xs ys)
  (bindrec ((prod (fun (zs)
                    (if (empty? zs)
                        #e
                        (bind x (head zs)
                          (bindrec ((map-duple (fun (ws)
                                                 (if (empty? ws)
                                                     #e
                                                     (prep (list x (head ws))
                                                           (map-duple (tail ws)))))))
                            (append (map-duple ys) ; Assume APPEND defined elsewhere
                                    (prod (tail zs)))))))))
    (prod xs)))
```

The same program can be expressed without block structure by passing an extra list argument `ys` to the `prod` function and an extra `x` argument to the `map-duple` function:

```
(def (cartesian-product-no-bs xs ys)
  (prod xs ys))

(def (prod zs ys)
  (if (empty? zs)
      #e
      (bind x (head zs)
        (append (map-duple ys x)
                (prod (tail zs) ys)))))

(def (map-duple ws x)
  (if (empty? ws)
      #e
      (prep (list x (head ws))
            (map-duple (tail ws) x))))
```

The ability to refer to names in enclosing scopes without passing them explicitly as parameters is a key advantage of block structure. This advantage may not seem so important in the context of simple examples like those above. A much more convincing example of the importance of block structure is the local definitions of the functions `eval` and `apply` within the OCAML `run` function in Fig. 2. In the block-structured version, `eval` takes only a single argument `venv`. Without block structure, it would be necessary for `eval` to take *three* arguments: `venv`, `fenv`, and `genv`. Changing every invocation of `eval` to pass two extra arguments would make it significantly more complex and harder to understand without changing its meaning.

Another advantage of block structure is that it helps to indicate which functions are used where in a program. A locally defined function that is not used in a first-class way can only be used in

the region of the program delimited by scope in which it is created.

2.2 The Syntax of FOBS

FOBS adds block structure to FOFL via local recursive function declarations that have the following form:

(funrec E_{body} $FD_1 \ldots FD_k$)

As in FOFL, each FOBS function declaration FD has the form

(def (F_{name} I_{formal_1} ... I_{formal_n}) E_{body})

As in FOFL, FOBS functions are second-class, and function names are in a different namespace from values. The function declarations in a `funrec` are mutually recursive; any function in the `funrec` may call any other function in the `funrec` in its body. The result of a `funrec` expression is the result of evaluating the final body expression E_{body} in a context where all the functions declared in the `funrec` are in scope.

The grammar of FOBS is exactly the same as the grammar of FOFL except for the addition of the the `funrec` expression. Unlike FOFL, FOBS does not need to handle top-level function declarations specially, since these can be desugared into `funrec`.

Here is a version of the cartesian product example expressed in FOBS:

```
(fobs (a b)
  (funrec (bindpar ((xs (range 1 a))
                    (ys (range 1 b)))
            (funrec (prod xs)
              (def (prod zs)
                (if (empty? zs)
                    #e
                    (bind x (head zs)
                      (append (funrec (map-duple ys)
                                (def (map-duple ws)
                                  (if (empty? ws)
                                      #e
                                      (prep (list x (head ws))
                                            (map-duple (tail ws))))))
                              (prod (tail zs))))))))))
    (def (append xs ys)
      (if (empty? xs)
          ys
          (prep (head xs)
                (append (tail xs) ys))))
    (def (range lo hi)
      (if (> lo hi)
          #e
          (prep lo (range (+ lo 1) hi))))))
```

2.3 Evaluation and Scope in FOBS

An environment model evaluator for `fobs` is presented in Fig. 3. The `env-run` function is parameterized over *two* scoping mechanisms: one for variable names (`vscope`) and one for function names (`fscope`). Each of these two scopes can independently be chosen to be static or dynamic via the following functions:

```
let static = fun senv denv -> senv
let dynamic = fun senv denv -> denv
```

```
(* Model a Fobs scoping mechanism as a way to combine static and dynamic environments *)
type 'a scoping = 'a Env.env (* static *) * 'a Env.env (* dynamic *) -> 'a Env.env
type closure = Clo of fcn * valu Env.env * closure Env.env (* function closures *)

(* val run : scoping -> scoping -> Fobs.pgm -> int list -> valu *)
(* vscope is variable scope and fscope is function scope *)
let rec run vscope fscope (Pgm(fmls,body)) ints =
  let flen = length fmls and ilen = length ints
  in if flen <> ilen then
       raise (EvalError ("Program expected " ^ (string_of_int flen)
                         ^ " arguments but got " ^ (string_of_int ilen)))
     else
       let rec eval exp venv (* current var. env. *) fenv (* current fun. env. *) =
         match exp with
           Lit v -> v
         | Var name ->
             (match Env.lookup name venv with
               Some(i) -> i
             | None -> raise (EvalError("Unbound variable: " ^ name)))
         | PrimApp(op, rands) -> (primopFunction op) (evalExps rands venv fenv)
         | If(tst, thn, els) ->
             (match eval tst venv fenv with
               Bool b -> if b then eval thn venv fenv else eval els venv fenv
             | v -> raise (EvalError ("Non-boolean test value " ^ (valuToString v)
                                      ^ " in if expression")))
         | Bind(name, defn, body) ->
             eval body (Env.bind name (eval defn venv fenv) venv) fenv
         | App(fname, rands) -> apply fname (evalExps rands venv fenv) venv fenv
         | Funrec(body, fcns) ->
             eval body venv
                  (Env.fix
                   (fun fe -> Env.bindAllThunks (map fcnName fcns)
                                                (map (fun fcn ->
                                                        (fun () -> Clo(fcn, venv, fe)))
                                                     fcns)
                                                fenv))
       and evalExps exps venv fenv = map (fun e -> eval e venv fenv) exps
       and apply fname actuals dvenv dfenv =
         match Env.lookup fname dfenv with
           None -> raise (EvalError ("unknown function " ^ fname))
         | Some (Clo(Fcn(name,formals,body), svenv, sfenv)) ->
             let flen = length formals and alen = length actuals
             in if flen <> alen then
               raise (EvalError ("Function " ^ name ^ " expected "
                                 ^ (string_of_int flen) ^ " arguments but got "
                                 ^ (string_of_int alen)))
             else eval body (Env.bindAll formals actuals (vscope svenv dvenv))
                            (fscope sfenv dfenv)
       in eval body
                (Env.make fmls (map (fun i -> Int i) ints)) (* initial venv *)
                Env.empty (* initial fenv *)
```

Figure 3: An environment model evaluator for FOBS parameterized over two scoping mechanisms: one (`fscope`) for the function namespace and one (`vscope`) for the value namespace.

We can also model "weird" scoping mechanisms if we so desire:

```
let empty = fun senv denv -> Env.empty
let merged1 = fun senv denv -> Env.merge senv denv
let merged2 = fun senv denv -> Env.merge denv senv
```

Similar to HOFL, static scoping in FOBS requires that a function be represented via a closure that associates a function with the environment in which it was created. A FOBS closure must be closed over *both* the variable environment and function environment in which it was created.

As examples of scoping in FOBS, consider the result of running each of the four programs in Fig. 4 on the argument list `[2; 1; 4]` using each possible combination of static and dynamic scope for `vscope` and `fscope`.

```
; SUM1: sums multiples of N between LO and HI
(fobs (n lo hi)
  (funrec (sum-loop lo 0)
    (def (multiple? x) (= 0 (rem x)))
    (def (rem y) (% y n))
    (def (sum-loop i sum)
           (if (> i hi)
               sum
               (sum-loop (+ i 1)
                         (if (multiple? i) (+ i sum) sum))))))

; SUM2: renames X in MULTIPLE? to N
(fobs (n lo hi)
  (funrec (sum-loop lo 0)
    (def (multiple? n) (= 0 (rem n)))
    (def (rem y) (% y n))
    (def (sum-loop i sum)
           (if (> i hi)
               sum
               (sum-loop (+ i 1)
                         (if (multiple? i) (+ i sum) sum))))))

; SUM3: adds nested FUNREC to SUM1
(fobs (n lo hi)
  (funrec (sum-loop lo 0)
    (def (multiple? x) (= 0 (rem x)))
    (def (rem y) (% y n))
    (def (sum-loop i sum)
      (funrec (if (> i  hi)
                  sum
                  (sum-loop (+ i 1) (new-sum i)))
        (def (new-sum z) (if (multiple? z) (inc z) sum))
        (def (inc w) (+ sum w))))))

; SUM4: renames Z to N in SUM3
(fobs (n lo hi)
  (funrec (sum-loop lo 0)
    (def (multiple? x) (= 0 (rem x)))
    (def (rem y) (% y n))
    (def (sum-loop i sum)
      (funrec (if (> i  hi)
                  sum
                  (sum-loop (+ i 1) (new-sum i)))
        (def (new-sum n) (if (multiple? n) (inc n) sum))
        (def (inc w) (+ sum w))))))

; Program 4 (renames INC to REM in Program 4)
(fobs (n lo hi)
  (funrec (sum-loop lo 0)
    (def (multiple? x) (= 0 (rem x)))
    (def (rem y) (% y n))
    (def (sum-loop i sum)
      (funrec (if (> i  hi)
                  sum
                  (sum-loop (+ i 1) (new-sum i)))
        (def (new-sum z) (if (multiple? z) (rem z) sum))
        (def (rem w) (+ sum w))))))
```

Figure 4: Examples illustrating scoping for both variable and function environments in Fobs.