CS251 Programming Languages **Handout # 23'**
Prof. Lyn Turbak **March 6, 2002**
Wellesley College

FOFL and FOBS

We have seen the power of first-class higher-order functions in Scheme, and have seen how they can be implemented in the toy HOFL language. The functions in most real programming languages (including C, Pascal, and Fortran) are much more limited than those in Scheme and HOFL. In this handout, we explore two toy languages with more limited kinds of functions:

1. FOFL (First-Order Functional Language) extends IBEX 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].

1 FOFL

1.1 Syntax of FOFL

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

- *Global Function Declarations* In addition to the program parameters and body expression, FOFL programs include arbitrarily many global function declarations that have the form `(fun` F_{name} `(`$I_{formal_1} \ldots I_{formal_n}$`)` E_{body}`)`.
- *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.
- Like HOFL, FOFL supports pairs (`pair`, `fst`,`snd`) and lists (`prepend`, `head`, `tail`, `empty`, and `empty?`)

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

```
(program (x)
  (fun fact (n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
  (fact x)))
```

[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.

$P \in \textit{Program}$	
$P \rightarrow$ (program $(I_{formal_1} \ldots I_{formal_n})$ $FD_1 \ldots FD_k$ E_{body})	Program
$FD \in \textit{Function Declaration}$	
$FD \rightarrow$ (fun F_{name} $(I_{formal_1} \ldots I_{formal_n})$ E_{body})	Function Declaration
$E \in \textit{Expression}$	
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:	
$E \rightarrow$ (scand E_1 E_2)	Short-Circuit And
$E \rightarrow$ (scor E_1 E_2)	Short-Circuit Or
$E \rightarrow$ (cond (E_{test_1} E_{body_1}) $\ldots$(E_{test_n} E_{body_n}) (else $E_{default}$))	Multi-branch Conditional
$E \rightarrow$ (bindseq ((I_{name_1} E_{defn_1}) $\ldots$(I_{name_n} E_{defn_n})) E_{body})	Sequential Binding
$E \rightarrow$ (bindpar ((I_{name_1} E_{defn_1}) $\ldots$(I_{name_n} E_{defn_n})) E_{body})	Parallel Binding
$L \in \textit{Literal}$	
$L \rightarrow N$	Numeric Literal
$L \rightarrow B$	Boolean Literal
$L \rightarrow Y$	Symbolic Literal

$O \in \textit{Primitive Operator}$: e.g., `+`, `<=`, `band`, `not`, `prepend`
$F \in \textit{Function Name}$: e.g., `f`, `sqr`, `+-and-*`
$I \in \textit{Identifier}$: e.g., `a`, `captain`, `fib_n-2`
$N \in \textit{Integer}$: e.g., `3`, `-17`
$B \in \textit{Boolean}$: `true` and `false`
$Y \in \textit{Symbol}$: e.g., `(symbol a)`, `(symbol captain)`, `(symbol fib_n-2)`

Figure 1: Grammar for the FOFL langauge.

```
(program (x)
  (fun fib (n)
    (if (<= n 1)
        n
        (+ (fib (- n 1))
           (fib (- n 2)))))
  (fib x))
```

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

```
(program (x)
  (fun even? (n)
    (if (= n 0)
        true
        (odd? (- n 1))))
  (fun odd? (n)
    (if (= n 0)
        false
        (even? (- n 1))))
  (even? x)))
```

Here is a simple example of list processing in FOFL:

```
(program (a b)
  (fun sum (lst)
    (if (empty? lst)
        0
        (+ (head lst)
           (sum (tail lst)))))
  (fun from-to (lo hi)
    (if (> lo hi)
        (empty)
        (prepend lo (from-to (+ lo 1) hi))))
  (sum (from-to a b)))
```

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 function declaration.

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 C, Pascal, and Common Lisp, as well as the toy languages FOFL and FOBS that we are studying. In contrast, many functional languages, such as Scheme, ML, and Haskell (as well as the toy HOFL language) 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:

```
(fun 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, Scheme, or ML, 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. As seen in the `funapp` clause, function names are looked up here.
  2. The **value environment** `venv` represents the value namespace that binds program parameters, function parameters, and `bind` names. As seen in the `varref` clause, variables are looked up here.
  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. Here are three different instantiations of `env-run`:

```
;; Global scoping is the equivalent of static scoping in FOFL.
;; The parent environment of an application frame is the global env.
(define fofl-run-global
  (env-run (lambda (genv venv) genv)))

;; Dynamic scoping is as in HOFL.
;; The parent environment of an application frame is
;; the current value environment
(define fofl-run-dynamic
  (env-run (lambda (genv venv) venv)))

;; Flat scoping is a possible, but not very useful, scoping mechanism.
;; The parent environment of an application frame is an empty environment.
(define fofl-run-flat
  (env-run (lambda (genv venv) (env-empty))))
```

```
(define (env-run scope)
  (lambda (pgm ints)
    (let ((dpgm (desugar-program pgm)))
      (let (;; FENV = global function environment
            (fenv (env-make (map fundecl-name (program-fundecls dpgm))
                            (program-fundecls dpgm)))
            ;; GENV = global value environment
            (genv (env-make (program-params dpgm) ints)))
        (define (env-eval exp venv)
          ;; Evaluate EXP relative to the (explicit) value environment VENV
          ;; and (implicit) function environment FENV and global environment GENV.
          (cond
            ((literal? exp) (literal-value exp))
            ((varref? exp)
             (let ((probe (env-lookup (varref-name exp) venv)))
               (if (unbound? probe)
                   (throw 'unbound-variable (varref-name exp))
                   probe)))
            ((primapp? exp)
             (primapply (primapp-rator exp)
                        (map (lambda (rand) (env-eval rand venv))
                             (primapp-rands exp))))
            ((bind? exp)
             (env-eval (bind-body exp)
                       (env-bind (bind-name exp)
                                 (env-eval (bind-defn exp) venv)
                                 venv)))
            ((if? exp)
             (let ((test (env-eval (if-test exp) venv)))
               (if (not (mini-boolean? test))
                   (throw 'if:non-boolean-test
                          (make-if test (if-then exp) (if-else exp)))
                   (if (mini-bool-to-scheme-bool test)
                       (env-eval (if-then exp) venv)
                       (env-eval (if-else exp) venv)))))
            ((funapp? exp)
             (let ((fundecl (env-lookup (funapp-rator exp) fenv))
                   (actuals (map (lambda (rand) (env-eval rand venv))
                                 (funapp-rands exp))))
               (if (unbound? fundecl)
                   (throw 'unknown-function (funapp-rator exp))
                   (funapply fundecl actuals venv))))
            (else (throw 'unknown-expression exp))
            ))
          (define (funapply fundecl actuals venv)
            ;; Apply FUNDECL function to ACTUALS in call environment VENV
            (env-eval (fundecl-body fundecl)
                      (env-extend (fundecl-params fundecl)
                                  actuals
                                  (scope genv venv))))
          ;; BODY OF (let ((fenv ...) (genv ...)) ...)
          (env-eval (program-body dpgm) genv)
          ))))
```

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

As an example of scoping in FOFL, consider the following program:

```
(program (a)
  (fun add-a (x) (+ x a))
  (fun try (a) (add-a (* 2 a)))
  (try 100))
```

2 FOBS

FOBS extends FOFL with **block structure** – the ability to define functions within other functions. Here we first explore block structure in the context of Scheme, and then discuss block structure in FOFL.

2.1 Block Structure in Scheme

Block structure in Scheme is realized via two constructs:

- Scheme's `lambda` construct allows creating functions anywhere in a program, even within other functions.
- Scheme's `letrec` construct allows creating collections of mutually recursive functions anywhere in a program, even within other functions.

As a simple example of block structure, consider the following program:

```
(define index-of-bs
  (lambda (elt lst)
    (letrec ((index-loop (lambda (i L)
                           (if (null? L)
                               -1
                               (if (eqv? elt (car L))
                                   i
                                   (index-loop (+ i 1) (cdr L)))))))
      (index-loop 1 lst))))
```

Note how the local `index-loop` function can refer to the `elt` of the enclosing `lambda` even though it is not passed as an explicit parameter.

Because `letrec` can be hard to read, Scheme supports syntactic sugar in which "internal" `define` constructs are desugared into `letrec`. For instance, the following is equivalent to the above:

```
(define index-of-bs
  (lambda (elt lst)
    (define index-loop
      (lambda (i L)
        (if (null? L)
            -1
            (if (eqv? elt (car L))
                i
                (index-loop (+ i 1) (cdr L))))))
    (index-loop 1 lst)))
```

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

```
(define index-of-no-bs
  (lambda (elt lst)
    (index-loop 1 lst elt)))

(define index-loop
  (lambda (i L elt)
    (if (null? L)
        -1
        (if (eqv? elt (car L))
            i
            (index-loop (+ i 1) (cdr L) elt)))))
```

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

```
(define cartesian-product-bs
  (lambda (lst1 lst2)
    (define prod
      (lambda (lst)
        (if (null? lst)
            '()
            (let ((elt (car lst)))
              (define map-duple
                (lambda (L)
                  (if (null? L)
                      '()
                      (cons (list elt (car L))
                            (map-duple (cdr L))))))
              (append (map-duple lst2)
                      (prod (cdr lst)))))))
    (prod lst1)))
```

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

```
(define cartesian-product-no-bs
  (lambda (lst1 lst2)
    (prod lst1 lst2)))

(define prod
  (lambda (lst1 lst2)
    (if (null? lst1)
        '()
        (let ((elt (car lst1)))
          (append (map-duple lst2 elt)
                  (prod (cdr lst1) lst2))))))

(define map-duple
  (lambda (L elt)
    (if (null? L)
        '()
        (cons (list elt (car L))
              (map-duple (cdr L) elt)))))
```

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 importants of block structure is the definition of the function `env-eval` within Fig. 2. In the block-structured version, `env-eval` takes only a single argument `venv`. Without block structure, it would be necessary for cdenv-eval to take *three* arguments: `venv`, `fenv`, and `genv`. Changing every invocation of `env-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 `letrec` 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:

$$\begin{array}{l}
\texttt{(funrec ((}F_{name_1}\ \texttt{(}I_{formal_{1,1}}\ \ldots\ I_{formal_{1,k_1}}\texttt{)}\ E_{body_1}\texttt{)} \\
\qquad\qquad\vdots \\
\qquad\qquad\texttt{(}F_{name_n}\ \texttt{(}I_{formal_{1,1}}\ \ldots\ I_{formal_{1,kn}}\texttt{)}\ E_{body_n}\texttt{))} \\
\quad E_{body}\texttt{)}
\end{array}$$

Each clause of the form

$$\texttt{(}F_{name_i}\ \texttt{(}I_{formal_{i,1}}\ \ldots\ I_{formal_{1,k_i}}\texttt{)}\ E_{body_i}\texttt{)}$$

declares a function named F_{name}i with formal parameters $I_{formal_{i,1}} \ldots I_{formal_{1,k_i}}$ and body E_{body_i}. 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:

```
(program (a b)
  (funrec
    ((from-to (lo hi)
       (if (> lo hi)
           (empty)
           (prepend lo (from-to (+ lo 1) hi))))
     (append (L1 L2)
       (if (empty? L1)
           L2
           (prepend (head L1)
                    (append (tail L1) L2)))))
    (bindpar ((lst1 (from-to 1 a))
              (lst2 (from-to 1 b)))
      (funrec
        ((prod (lst)
           (if (empty? lst)
               (empty)
               (bind elt (head lst)
                 (funrec
                   ((map-duple (L)
                      (if (empty? L)
                          (empty)
                          (prepend (pair elt (head L))
                                   (map-duple (tail L))))))
                   (append (map-duple lst2)
                           (prod (tail lst))))))))
        (prod lst1)))))
```

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 function names (`fscope`) and one for value names (`vscope`). As shown below, each of these two scopes can independently be chosen to be static or dynamic:

```
(define (env-run fscope vscope)
  (lambda (pgm args)
    (define (env-eval exp fenv venv)
      ;; Evaluate EXP in function env. FENV and value env. VENV.
      (cond
        ((literal? exp) (literal-value exp))
        ((varref? exp)
         (let ((probe (env-lookup (varref-name exp) venv)))
           (if (unbound? probe)
               (throw 'unbound-variable (varref-name exp))
               probe)))
        ((primapp? exp)
         (primapply (primapp-rator exp)
                    (map (lambda (rand) (env-eval rand fenv venv))
                         (primapp-rands exp))))
        ((bind? exp)
         (env-eval (bind-body exp)
                   fenv
                   (env-bind (bind-name exp)
                             (env-eval (bind-defn exp) fenv venv)
                             venv)))
        ((if? exp)
         (let ((test (env-eval (if-test exp) fenv venv)))
           (if (not (mini-boolean? test))
               (throw 'if:non-boolean-test (make-if test (if-then exp) (if-else exp)))
               (if (mini-bool-to-scheme-bool test)
                   (env-eval (if-then exp) fenv venv)
                   (env-eval (if-else exp) fenv venv)))))
        ((funapp? exp)
         (let ((closure (env-lookup (funapp-rator exp) fenv))
               (actuals (map (lambda (rand) (env-eval rand fenv venv))
                             (funapp-rands exp))))
           (if (unbound? closure)
               (throw 'unknown-function (funapp-rator exp))
               (funapply closure actuals fenv venv))))
        ((funrec? exp)
         (env-eval (funrec-body exp)
                   (function-env-extend (funrec-fundecls exp) fenv venv)
                   venv))
        (else (throw 'unknown-expression exp))))
    (define (funapply closure actuals dyn-fenv dyn-venv)
      ;; Apply CLOSURE to ACTUALS relative to dynamic function and value envs
      (env-eval (closure-body closure)
                (fscope dyn-fenv (closure-fenv closure))
                (env-extend (closure-formals closure)
                            actuals
                            (vscope dyn-venv (closure-venv closure)))))
    ;; Body of (lambda (pgm args) ...)
    (env-eval (desugar (program-body pgm))
              (env-empty)                       ; initial function env
              (env-extend (program-formals pgm) ; initial value env
                          args
                          (env-empty)))))
```

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.

```
(define dynamic (lambda (dynamic-env static-env) dynamic-env))
(define static (lambda (dynamic-env static-env) static-env))

;; Both function and variable namespaces are statically scoped
(define fobs-run-statfun-statvar (env-run static static))

;; Functions are statically scoped; variables dynamically scoped
(define fobs-run-statfun-dynvar (env-run static dynamic))

;; Functions are dynamically scoped; variables statically scoped
(define fobs-run-dynfun-statvar (env-run dynamic static))

;; Both function and variable namespaces are dynamically scoped
(define fobs-run-dynfun-dynvar (env-run dynamic dynamic))
```

Similar to HOFL, static scoping in FOBS requires that a function be represented via a closure that associates a function with both the function environment and value environment in which it was created.

As examples of scoping in FOBS, below we consider four versions of the same program. We shall consider running each program on the arguments `(3 1 10)`.

```
;; Version 1: no block structure
(program (n lo hi)
  (funrec ((multiple? (x) (zero? (mod-n x)))
           (mod-n (y) (mod y n))
           (zero? (a) (= a 0))
           (done? (x) (> x hi))
           (sum-loop (cur sum)
             (if (done? cur)
                 sum
                 (sum-loop (+ cur 1)
                           (if (multiple? cur)
                               (+ sum cur)
                               sum)))))
    (sum-loop lo 0)))
```

```
;; Version 2: with block structure
(program (n lo hi)
  (funrec ((multiple? (x) (zero? (mod-n x)))
           (mod-n (y) (mod y n))
           (zero? (a) (= a 0))
           (done? (x) (> x hi))
           (sum-loop (cur sum)
             (funrec ((new-sum (c)
                        (if (multiple? c) (inc-sum c) sum))
                      (inc-sum (y) (+ sum y)))
               (if (done? cur)
                   sum
                   (sum-loop (+ cur 1) (new-sum cur))))))
    (sum-loop lo 0)))


;; Version 3: C renamed to N
(program (n lo hi)
  (funrec ((multiple? (x) (zero? (mod-n x)))
           (mod-n (y) (mod y n))
           (zero? (a) (= a 0))
           (done? (x) (> x hi))
           (sum-loop (cur sum)
             (funrec ((new-sum (n)
                        (if (multiple? n) (inc-sum n) sum))
                      (inc-sum (y) (+ sum y)))
               (if (done? cur)
                   sum
                   (sum-loop (+ cur 1) (new-sum cur))))))
    (sum-loop lo 0)))


;; Version 4: INC-SUM renamed to MOD-N
(program (n lo hi)
  (funrec ((multiple? (x) (zero? (mod-n x)))
           (mod-n (y) (mod y n))
           (zero? (a) (= a 0))
           (done? (x) (> x hi))
           (sum-loop (cur sum)
             (funrec ((new-sum (c)
                        (if (multiple? c) (mod-n c) sum))
                      (mod-n (y) (+ sum y)))
               (if (done? cur)
                   sum
                   (sum-loop (+ cur 1) (new-sum cur))))))
    (sum-loop lo 0)))
```