How to Have Fun

Barış Aktemur
CS321 Programming Languages
Ozyegin University

We have been designing a language called Deve for which we are also implementing an interpreter. At this point we have many features enabled in Deve, except for one big ticket item: functions. In this lecture, we will see how we can extend Deve with functions. We will build our code on top of Deve 2.0, available at `https://github.com/aktemur/cs321/tree/master/Deve-2.0`

Nameless Functions

The first thing we do is to add the ability to define nameless function as in

```
fun x -> x + 5
```

The new grammar rule is

```
exp  ::= ...
       | FUN NAME ARROW exp
```

A function definition has a parameter (a `string` value), and the function body (an `exp`). We are going to represent a function definition in the AST using the following `Fun` constructor:

```
type exp = CstI of int
           ...
         | Fun of string * exp
```

In terms the precedence/associativity/ambiguity a function definition works exactly like let-in, if-then-else, or match-with; its left-hand-side is "closed" with the `FUN` token, whereas the right-hand-side is "open" with an `exp` non-terminal. Therefore, we handle parsing just like we handle other Level-1 expressions.

For the lexer, just add a new token and a case to recognize "fun" as a keyword. The parser is as follows:

```
and parseLevel1ExpOrOther otherParseFun tokens =
  match tokens with
  | LET::rest -> let (e, tokens2) = parseLetIn tokens
                 in (e, tokens2)
  | IF::rest  -> let (e, tokens2) = parseIfThenElse tokens
                 in (e, tokens2)
  | MATCH::rest -> let (e, tokens2) = parseMatchPair tokens
                   in (e, tokens2)
  | FUN::rest -> let (e, tokens2) = parseFun tokens
                 in (e, tokens2)
  | _         -> let (e, tokens2) = otherParseFun tokens
```

```
                in (e, tokens2)
...
and parseFun tokens =
  match tokens with
  | FUN::NAME(x)::ARROW::rest ->
     let (e, tokens1) = parseExp rest in
     (Fun(x, e), tokens1)
  | _ -> failwith "Badly formed fun definition."
```

Let's check the parser:

```
# parse "fun x -> 4";;
- : exp = Fun ("x", CstI 4)
# parse "fun x -> x + 5";;
- : exp = Fun ("x", Binary ("+", Var "x", CstI 5))
# parse "let f = fun x -> x + 5 in f";;
- : exp = LetIn ("f", Fun ("x", Binary ("+", Var "x", CstI 5)), Var "f")
```

Next step is the evaluator. Remember from the beginning of the semester that functions are represented with values called "closures" (see OCaml notes, part 02). A closure is a tuple that contains the function parameter, function body, and the environment in which the function was defined. A function definition immediately reduces to a closure, by freezing its body an its environment. (The body is evaluated at the time of application, not definition.) So, here is the evaluator:

```
type value = ...
  | Closure of string * exp * ((string * value) list)  (* parameter, body, environment *)

let rec eval e env =
  match e with
  ...
  | Fun(x, e) -> Closure(x, e, env)
```

Let's test:

```
# run "fun x -> 42";;
- : value = Closure ("x", CstI 42, [])
# run "let y = 2+3 in fun x -> 42";;
- : value = Closure ("x", CstI 42, [("y", Int 5)])
# run "let f = fun x -> 42 in f";;
- : value = Closure ("x", CstI 42, [])
```

Good. We are now able to define functions, but we can't use them, yet, because we don't have function application. Before we talk about function application, let us improve the parser so that we can define functions with a neater syntax, such as

```
  let f x = x + 6 in ...
```

instead of

```
  let f = fun x -> x + 6 in ...
```

Note that the former form is just a syntactic sugaring (i.e. convenience) that is exactly the same as the latter. In general, any let-binding of a function such as

```
let f x y z = ..body..
in ...
```

can be written as follows:

```
let f = fun x -> fun y -> fun z -> ..body..
in ...
```

To handle the let-binding for function definitions, we add the following grammar rule (I'll only do the case with one parameter, I leave multi-parameter case as an exercise to you):

```
exp  ::= ...
     | LET NAME NAME EQUALS exp IN exp
```

We extend the parser definition as below. Note that no change is needed in the evaluator, because the parser converts a let-binding of a function to a nameless function definition.

```
and parseLetIn tokens =
  match tokens with
  | LET::NAME(x)::EQUALS::rest ->
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetIn(x, e1, e2), tokens3)
  | LET::NAME(f)::NAME(x)::EQUALS::rest ->      (* NEW CASE *)
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetIn(f, Fun(x, e1), e2), tokens3)
  | _ -> failwith "Should not be possible."
```

And testing...

```
# parse "let f x = 42 in f";;
- : exp = LetIn ("f", Fun ("x", CstI 42), Var "f")
# parse "let y = 5
         in let f x = x + y
            in f";;
- : exp = LetIn ("y", CstI 5,
             LetIn ("f", Fun ("x", Binary ("+", Var "x", Var "y")), Var "f"))
# run "let x = 12
       in let f y = x + y
          in let x = 99
             in f";;
- : value = Closure ("y", Binary ("+", Var "x", Var "y"), [("x", Int 12)])
```

Function Application

Now that we can *define* function, let's *apply* them. The grammar for function application is this:

```
exp  ::= ...
     | exp exp
```

There are no terminals here! When two expressions are written side by side, it means function application. The expression on the left is the function, and the one on the right is the argument. To represent this, we will add the AST constructor shown below (we could have used `Binary` as well, but function application is so important that I think it deserves its own constructor):

```
type exp = ...
         | App of exp * exp
```

Let's now focus on the parser. The application rule is liable to great many ambiguities. For instance, can you come up with two different parse trees for "x y z"? How about "f n+1"? And "let x = 5 in f x"? We will resolve ambiguity based in the following rules:

- Function application has the highest precedence.
- Function application associates to the left.

We will attack associativity similar to the tactic we used for binary operators: a helper function that takes the accumulated left-hand-side expression. We will handle precedence by covering the application rule at level-4. Here is the code:

```
and parseLevel4Exp tokens =
  let rightHandSideExists token =
    match token with
    (* tokens that are the beginning of an atom *)
    | INT _ | NAME _ | BOOL _ | LPAR | FST | SND | NOT -> true
    (* other tokens that may start the right-hand-side exp *)
    | LET | IF | MATCH | FUN -> true
    | _ -> false
  in
  let rec helper tokens e1 =
    match tokens with
    | tok::rest when rightHandSideExists(tok) ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseAtomExp tokens
       in helper tokens2 (App(e1, e2))
    | _ ->
       (e1, tokens)
  in let (e1, tokens1) = parseAtomExp tokens in
     helper tokens1 e1

and parseAtomExp tokens =
  match tokens with
  | INT i :: rest -> (CstI i, rest)
  | NAME x :: rest -> (Var x, rest)
  | BOOL b :: rest -> (CstB b, rest)
  ...
```

Here, we use an auxiliary function named `rightHandSideExists` to check whether the next token could start the right-hand-side expression of an application. If it can, we continue, otherwise we stop parsing the application. Let's test.

```
# parse "x y z";;
- : exp = App (App (Var "x", Var "y"), Var "z")
# parse "f 3 + 1";;
- : exp = Binary ("+", App (Var "f", CstI 3), CstI 1)
# parse "3 + f 1";;
- : exp = Binary ("+", CstI 3, App (Var "f", CstI 1))
# parse "let x = 5 in f x";;
- : exp = LetIn ("x", CstI 5, App (Var "f", Var "x"))
# parse "let x = f 4 in 42";;
- : exp = LetIn ("x", App (Var "f", CstI 4), CstI 42)
# parse "let x = 5 in 3 + f 1";;
- : exp = LetIn ("x", CstI 5, Binary ("+", CstI 3, App (Var "f", CstI 1)))
# parse "f let x = 5 in x";;
- : exp = App (Var "f", LetIn ("x", CstI 5, Var "x"))
```

I'm relieved that we are done with the parser. That was a hard case! Let us now focus on the evaluator. We need to add an `App` case to the `eval` function. For this, again remember from the beginning of the semester (OCaml notes, part 02) that a function application $e_1\ e_2$ is evaluated as follows:

- Suppose the function application is being evaluated in environment ρ.
- Evaluate the left-hand-side expression e_1 in ρ. This should give a closure, $\langle x, e_b, \rho_f \rangle$, where x is the function parameter, e_b is the function body, and ρ_f is the environment that was frozen inside the closure.
- Evaluate the right-hand-side expression e_2 in ρ. This will give us a value v. This is the argument of the function.
- Next, we evaluate the function body by passing the argument in place of the parameter. We do that by binding v to x and putting that on top of ρ_f, and then evaluating e_b in this new environment.

Here is the code:

```
let rec eval e env =
  match e with
  ...
  | App(e1, e2) ->
     let closure = eval e1 env in
     (match closure with
      | Closure(x, funBody, funEnv) ->
         let v2 = eval e2 env
         in eval funBody ((x, v2)::funEnv)
      | _ -> failwith "Application wants to see a closure!"
     )
```

And the test cases:

```
# run "let f x = 42 in f 9";;
- : value = Int 42
```

```
# run "let f x = x + 30 in f 12";;
- : value = Int 42
# run "let f = fun x -> fun y -> x + y
       in f 30 12";;
- : value = Int 42
# run "let x = 12
       in let f y = x + y
          in let x = 99
             in f (x + 1)";;
- : value = Int 112
# run "let f = fun x -> fun y -> x + y
       in f 30";;
- : value = Closure ("y", Binary ("+", Var "x", Var "y"), [("x", Int 30)])
```

Recursion

In OCaml, recursive functions are defined using the `let rec` binding. Let us now implement that. First, the grammar rule:

```
exp ::= ...
      | LET REC NAME NAME EQUALS exp IN exp
```

In the lexer, simply add a case to recognize `rec` as a keyword. To distinguish an ordinary let-binding from a let-rec, we add the following AST constructor:

```
  type exp = ...
           | LetRec of string * string * exp * exp
```

Parser is straightforward:

```
and parseLetIn tokens =
  match tokens with
  ...
  | LET::REC::NAME(f)::NAME(x)::EQUALS::rest ->   (* NEW CASE *)
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetRec(f, Fun(x, e1), e2), tokens3)
  | _ -> failwith "Badly formed let expression"
```

A simple test:

```
# parse "let rec f x = x in f 42";;
- : exp = LetRec ("f", "x", Var "x", App (Var "f", CstI 42))
```

Good. Let's now think about the evaluator. When I evaluate a `LetRec(f, x, e1, e2)`, I need to define a *recursive* function whose name is `f`, whose parameter is `x`, and whose body is `e1`. I will then put that information into the environment to continue evaluating `e2`. We currently have the `Closure` constructor to define function values, but that constructor does NOT have room for the function name, which we should not lose. So, I define another closure that I'll use for recursive functions. The definition of `eval` for the `LetRec` case is then straightforward:

```
type value = ...
           | Closure of string * exp * ((string * value) list)
           | RecClosure of string * string * exp * ((string * value) list)  (* NEW *)

let rec eval e env =
  match e with
  ...
  | LetRec(f, x, e1, e2) ->
     let closure = RecClosure(f, x, e1, env)
     in eval e2 ((f, closure)::env)
```

Now that we have a second kind of a closure, we need to fix function application case to handle `RecClosure` as well. As a start, let us implement this exactly the same as an ordinary closure:

```
let rec eval e env =
  match e with
  ...
  | App(e1, e2) ->
     let closure = eval e1 env in
     (match closure with
      | Closure(x, funBody, funEnv) ->
         let v2 = eval e2 env
         in eval funBody ((x, v2)::funEnv)
      | RecClosure(f, x, funBody, funEnv) ->      (* NEW CASE *)
         let v2 = eval e2 env
         in eval funBody ((x, v2)::funEnv)
      | _ -> failwith "Application wants to see a closure!"
     )
```

This would work for functions that do not contain recursive applications:

```
# run "let rec f x = x + 5 in f 37";;
- : value = Int 42
```

However, let's now define a recursive function and see what happens:

```
# run "let rec fact n =
         if n <= 0 then 1 else n * fact (n-1)
       in fact 5";;
Exception: Failure "Unbound name fact".
```

To understand the essence of the problem, let's check the closure we get for `fact`:

```
# run "let rec fact n =
         if n <= 0 then 1 else n * fact (n-1)
       in fact";;
- : value =
RecClosure("fact", "n",
  If(Binary("<=", Var "n", CstI 0),
    CstI 1,
    Binary ("*", Var "n", App (Var "fact", Binary ("-", Var "n", CstI 1)))),
  [])
```

The closure contains the empty environment `[]`. When evaluating the application `fact 5`, we will bind 5 to `n`, and put that on top of the empty environment to obtain `[y ↦ 5]`. It is this environment that is used to evaluate the function body. When we hit `Var "fact"`, we look up `"fact"` in `[y ↦ 5]`, and there where we fail. To fix this problem, right before we evaluate the function body, we also need to pass the information about the function itself. Luckily, this is easy:

```
let rec eval e env =
  match e with
  ...
  | App(e1, e2) ->
     let closure = eval e1 env in
     (match closure with
      | Closure(x, funBody, funEnv) ->
         let v2 = eval e2 env
         in eval funBody ((x, v2)::funEnv)
      | RecClosure(f, x, funBody, funEnv) ->
         let v2 = eval e2 env
         in eval funBody ((f,closure)::(x, v2)::funEnv)    (* CHANGED *)
      | _ -> failwith "Application wants to see a closure!"
     )
```

Cross your fingers, and test:

```
# run "let rec fact n =
         if n <= 0 then 1 else n * fact (n-1)
       in fact 5";;
- : value = Int 120
# run "let rec fact n =
         if n <= 0 then 1 else n * fact (n-1)
       in fact 6";;
- : value = Int 720
# run "let rec fib n =
         if n <= 0 then 1
         else if n <= 1 then 1
         else fib (n-1) + fib (n-2)
       in (fib 5, (fib 6, fib 7))";;
- : value = Pair (Int 8, Pair (Int 13, Int 21))
# run "let rec power x = fun n ->
         if n <= 0 then 1 else x * power x (n-1)
       in power 3 4";;
- : value = Int 81
```

Now the opportunities are endless...

Clean-up

At this point, we have all the necessary infrastructure to define the current unary operators `fst`, `snd`, and `not` as ordinary functions. So, to keep the core language small and tidy, I'll now simply throw away our definitions regarding `fst`, `snd`, and `not`. These will no longer be built-in operators, but can be defined and used as ordinary functions:

```
# run "let fst p = match p with (x,y) -> x in
       let snd p = match p with (x,y) -> y in
       let p = (3, 5) in fst p";;
- : value = Int 3
# run "let fst p = match p with (x,y) -> x in
       let snd p = match p with (x,y) -> y in
       let p = (3, 5) in snd p";;
- : value = Int 5
# run "let not b = if b then false else true
       in not (6 < 4)";;
- : value = Bool true
# run "let not b = if b then false else true
       in not true";;
- : value = Bool false
```

A better idea is to actually define these unary operators as part of a "standard library" that is loaded by default to each program, so that the programmer does not have to define them each time.

```
let wrapStdlib code =
  "let fst p = match p with (x,y) -> x in
```

```
   let snd p = match p with (x,y) -> y in
   let not b = if b then false else true
   in " ^ code
;;

let run code =
  eval (parse (wrapStdlib code)) []
;;
```

```
# run "fst (3, 4)";;
- : value = Int 3
# run "let p = (3, 4) in fst p + snd p";;
- : value = Int 7
# run "not (8 < 9)";;
- : value = Bool false
```

We are such nice language designers, the Deve programmers will adore us!

Appendix

At this point, we have reached a version of the language that we will call Deve 3.0. It is available at https://github.com/aktemur/cs321/tree/master/Deve-3.0. The final code is also shown below:

EVALUATOR:

```
type exp = CstI of int
         | CstB of bool
         | Var of string
         | Binary of string * exp * exp
         | LetIn of string * exp * exp
         | LetRec of string * string * exp * exp
         | If of exp * exp * exp
         | MatchPair of exp * string * string * exp
         | Fun of string * exp
         | App of exp * exp

type value = Int of int
           | Bool of bool
           | Pair of value * value
           | Closure of string * exp * ((string * value) list)  (* parameter, body, environment *)
           | RecClosure of string * string * exp * ((string * value) list) (* also contains the functio

let rec lookup x env =
  match env with
  | [] -> failwith ("Unbound name " ^ x)
  | (y,i)::rest -> if x = y then i
                   else lookup x rest

(* eval: exp -> (string * value) list -> value *)
let rec eval e env =
  match e with
  | CstI i -> Int i
  | CstB b -> Bool b
  | Var x -> lookup x env
  | Binary(op, e1, e2)  ->
     let v1 = eval e1 env in
     let v2 = eval e2 env in
     (match op, v1, v2 with
      | "+", Int i1, Int i2 -> Int(i1 + i2)
      | "-", Int i1, Int i2 -> Int(i1 - i2)
      | "*", Int i1, Int i2 -> Int(i1 * i2)
      | "/", Int i1, Int i2 -> Int(i1 / i2)
      | "<", Int i1, Int i2 -> Bool(i1 < i2)
      | "<=", Int i1, Int i2 -> Bool(i1 <= i2)
      | ",", _, _ -> Pair(v1, v2)
     )
  | LetIn(x, e1, e2) -> let v = eval e1 env
                        in let env' = (x, v)::env
```

```
                        in eval e2 env'
| LetRec(f, x, e1, e2) ->
   let closure = RecClosure(f, x, e1, env)
   in eval e2 ((f, closure)::env)
| If(e1, e2, e3) -> (match eval e1 env with
                     | Bool true -> eval e2 env
                     | Bool false -> eval e3 env
                     | _ -> failwith "Condition should be a Bool.")
| MatchPair(e1, x, y, e2) ->
   (match eval e1 env with
    | Pair(v1, v2) -> eval e2 ((x,v1)::(y,v2)::env)
    | _ -> failwith "Pair pattern matching works on pair values only (obviously)!"
   )
| Fun(x, e) -> Closure(x, e, env)
| App(e1, e2) ->
   let closure = eval e1 env in
   (match closure with
    | Closure(x, funBody, funEnv) ->
       let v2 = eval e2 env
       in eval funBody ((x, v2)::funEnv)
    | RecClosure(f, x, funBody, funEnv) ->
       let v2 = eval e2 env
       in eval funBody ((f,closure)::(x, v2)::funEnv)
    | _ -> failwith "Application wants to see a closure!"
   )
```

LEXER:

```
(* This is the lexer.
   The goal of the lexer is to take a string
   and recognize the tokens in it.
   A "token" is a categorized unit
   of input, such as "an integer", "the plus operator",
   "a name", etc.
 *)
type token = INT of int
           | BOOL of bool
           | NAME of string
           | PLUS | STAR | MINUS | SLASH
           | LESS | LESSEQ | GREATEREQ
           | LET | EQUALS | IN
           | IF | THEN | ELSE
           | LPAR | RPAR
           | COMMA
           | MATCH | WITH | ARROW
           | FUN | REC
           | ERROR of char
           | EOF
;;

let isDigit c = '0' <= c && c <= '9'

let digitToInt c = int_of_char c - int_of_char '0'

let isLowercaseLetter c = 'a' <= c && c <= 'z'

let isUppercaseLetter c = 'A' <= c && c <= 'Z'

let isLetter c = isLowercaseLetter c || isUppercaseLetter c

let charToString c = String.make 1 c

let keyword s =
  match s with
  | "let" -> LET
  | "in" -> IN
  | "if" -> IF
  | "then" -> THEN
  | "else" -> ELSE
  | "true" -> BOOL true
  | "false" -> BOOL false
  | "match" -> MATCH
  | "with" -> WITH
  | "fun" -> FUN
  | "rec" -> REC
  | _ -> NAME s
```

```
(*  tokenize: char list -> token list  *)
let rec tokenize chars =
  match chars with
  | [] -> [EOF]
  | '-'::'>'::rest -> ARROW::(tokenize rest)
  | '+'::rest -> PLUS::(tokenize rest)
  | '*'::rest -> STAR::(tokenize rest)
  | '-'::rest -> MINUS::(tokenize rest)
  | '/'::rest -> SLASH::(tokenize rest)
  | '='::rest -> EQUALS::(tokenize rest)
  | '<'::'='::rest -> LESSEQ::(tokenize rest)
  | '<'::rest -> LESS::(tokenize rest)
  | '>'::'='::rest -> GREATEREQ::(tokenize rest)
  | ' '::rest -> tokenize rest
  | '\t'::rest -> tokenize rest
  | '\n'::rest -> tokenize rest
  | '('::rest  -> LPAR::(tokenize rest)
  | ')'::rest  -> RPAR::(tokenize rest)
  | ','::rest  -> COMMA::(tokenize rest)
  | c::rest when isDigit(c) ->
     tokenizeInt rest (digitToInt c)
  | c::rest when isLowercaseLetter(c) ->
     tokenizeName rest (charToString c)
  | c::rest -> (ERROR c)::(tokenize rest)

and tokenizeInt chars n =
  match chars with
  | c::rest when isDigit(c) ->
     tokenizeInt rest (n * 10 + (digitToInt c))
  | _ -> (INT n)::(tokenize chars)

and tokenizeName chars s =
  match chars with
  | c::rest when isLetter(c) || isDigit(c) ->
     tokenizeName rest (s ^ (charToString c))
  | _ -> (keyword s)::(tokenize chars)
;;

let chars_of_string s =
  let rec helper n acc =
    if n = String.length s
    then List.rev acc
    else let c = String.get s n
         in helper (n+1) (c::acc)
  in helper 0 []
;;

let scan s =
  tokenize (chars_of_string s)
;;
```

PARSER:

```
(* A helper function to convert a token to a string *)
let toString tok =
  match tok with
  | INT i -> "INT(" ^ string_of_int i ^ ")"
  | BOOL b -> "BOOL(" ^ string_of_bool b ^ ")"
  | NAME x -> "NAME(\"" ^ x ^ "\")"
  | PLUS -> "PLUS"
  | STAR -> "STAR"
  | MINUS -> "MINUS"
  | SLASH -> "SLASH"
  | LET -> "LET"
  | EQUALS -> "EQUALS"
  | IN -> "IN"
  | IF -> "IF"
  | THEN -> "THEN"
  | ELSE -> "ELSE"
  | ERROR c -> "ERROR('" ^ (charToString c) ^ "')"
  | EOF -> "EOF"
  | LESS -> "LESS"
  | LESSEQ -> "LESSEQ"
  | GREATEREQ -> "GREATEREQ"
  | LPAR -> "LPAR"
  | RPAR -> "RPAR"
  | COMMA -> "COMMA"
  | MATCH -> "MATCH"
  | WITH -> "WITH"
  | ARROW -> "ARROW"
  | FUN -> "FUN"
  | REC -> "REC"

(* consume: token -> token list -> token list
   Enforces that the given token list's head is the given token;
   returns the tail.
*)
let consume tok tokens =
  match tokens with
  | [] -> failwith ("I was expecting to see a " ^ (toString tok))
  | t::rest when t = tok -> rest
  | t::rest -> failwith ("I was expecting a " ^ (toString tok) ^
                         ", but I found a " ^ toString(t))

(* parseExp: token list -> (exp, token list)
   Parses an exp out of the given token list,
   returns that exp together with the unconsumed tokens.
 *)
let rec parseExp tokens =
  parseLevel1ExpOrOther parseLevel1_5Exp tokens

and parseLevel1ExpOrOther otherParseFun tokens =
```

```
  match tokens with
  | LET::rest -> let (e, tokens2) = parseLetIn tokens
                 in (e, tokens2)
  | IF::rest  -> let (e, tokens2) = parseIfThenElse tokens
                 in (e, tokens2)
  | MATCH::rest -> let (e, tokens2) = parseMatchPair tokens
                   in (e, tokens2)
  | FUN::rest -> let (e, tokens2) = parseFun tokens
                 in (e, tokens2)
  | _         -> let (e, tokens2) = otherParseFun tokens
                 in (e, tokens2)

and parseLetIn tokens =
  match tokens with
  | LET::NAME(x)::EQUALS::rest ->
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetIn(x, e1, e2), tokens3)
  | LET::NAME(f)::NAME(x)::EQUALS::rest ->
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetIn(f, Fun(x, e1), e2), tokens3)
  | LET::REC::NAME(f)::NAME(x)::EQUALS::rest ->
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetRec(f, x, e1, e2), tokens3)
  | _ -> failwith "Badly formed let expression"

and parseIfThenElse tokens =
  let rest = consume IF tokens in
  let (e1, tokens1) = parseExp rest in
  let tokens2 = consume THEN tokens1 in
  let (e2, tokens3) = parseExp tokens2 in
  let tokens4 = consume ELSE tokens3 in
  let (e3, tokens5) = parseExp tokens4 in
  (If(e1, e2, e3), tokens5)

and parseMatchPair tokens =
  let rest = consume MATCH tokens in
  let (e1, tokens1) = parseExp rest in
  match tokens1 with
  | WITH::LPAR::NAME(x)::COMMA::NAME(y)::RPAR::ARROW::rest1 ->
     let (e2, tokens2) = parseExp rest1 in
     (MatchPair(e1, x, y, e2), tokens2)
  | _ -> failwith "Badly formed match expression."

and parseFun tokens =
  match tokens with
```

```
  | FUN::NAME(x)::ARROW::rest ->
     let (e, tokens1) = parseExp rest in
     (Fun(x, e), tokens1)
  | _ -> failwith "Badly formed fun definition."

and parseLevel1_5Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | LESS::rest ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseLevel2Exp rest
       in helper tokens2 (Binary("<", e1, e2))
    | LESSEQ::rest ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseLevel2Exp rest
       in helper tokens2 (Binary("<=", e1, e2))
    | GREATEREQ::rest ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseLevel2Exp rest
       in helper tokens2 (If(Binary("<", e1, e2), CstB false, CstB true))
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel2Exp tokens in
     helper tokens1 e1

and parseLevel2Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | PLUS::rest ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseLevel3Exp rest
       in helper tokens2 (Binary("+", e1, e2))
    | MINUS::rest ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseLevel3Exp rest
       in helper tokens2 (Binary("-", e1, e2))
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel3Exp tokens in
     helper tokens1 e1

and parseLevel3Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | STAR::rest ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseLevel4Exp rest
       in helper tokens2 (Binary("*", e1, e2))
    | SLASH::rest ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseLevel4Exp rest
       in helper tokens2 (Binary("/", e1, e2))
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel4Exp tokens in
     helper tokens1 e1

and parseLevel4Exp tokens =
  let rightHandSideExists token =
    match token with
    (* tokens that are the beginning of an atom *)
```

```
    | INT _ | NAME _ | BOOL _ | LPAR -> true
    (* other tokens that may start the right-hand-side exp *)
    | LET | IF | MATCH | FUN -> true
    | _ -> false
  in
  let rec helper tokens e1 =
    match tokens with
    | tok::rest when rightHandSideExists(tok) ->
       let (e2, tokens2) = parseLevel1ExpOrOther parseAtomExp tokens
       in helper tokens2 (App(e1, e2))
    | _ ->
       (e1, tokens)
  in let (e1, tokens1) = parseAtomExp tokens in
     helper tokens1 e1

and parseAtomExp tokens =
  match tokens with
  | INT i :: rest -> (CstI i, rest)
  | NAME x :: rest -> (Var x, rest)
  | BOOL b :: rest -> (CstB b, rest)
  | LPAR::rest ->
     let (e1, tokens1) = parseExp rest in
     (match tokens1 with
      | RPAR::rest1 -> (e1, rest1)
      | COMMA::rest1 ->
         let (e2, tokens2) = parseExp rest1 in
         let rest2 = consume RPAR tokens2 in
         (Binary(",", e1, e2), rest2)
      | _ -> failwith "Badly formed parethesized exp."
     )
  | t::rest -> failwith ("Unsupported token: " ^ toString(t))
  | [] -> failwith "No more tokens???"

(* parseMain: token list -> exp *)
let parseMain tokens =
  let (e, tokens1) = parseExp tokens in
  let tokens2 = consume EOF tokens1 in
  if tokens2 = [] then e
  else failwith "Oops."

(* parse: string -> exp *)
let rec parse s =
  parseMain (scan s)
```

MAIN:

```
#use "deve.ml";;
#use "lexer.ml";;
#use "parser.ml";;

open Printf

let wrapStdlib code =
  "let fst p = match p with (x,y) -> x in
   let snd p = match p with (x,y) -> y in
   let not b = if b then false else true
   in " ^ code
;;

let run code =
  eval (parse (wrapStdlib code)) []
;;

let runBare code =
  eval (parse code) []
;;

let rec valToString v =
  match v with
  | Int i -> string_of_int i
  | Bool b -> string_of_bool b
  | Pair(v1, v2) -> "(" ^ valToString(v1) ^ "," ^ valToString(v2) ^ ")"
  | Closure(x,e,env) -> "<fun>"
  | RecClosure(f,x,e,env) -> "<fun>"

let deveREPL() =
  let rec readInput() =
    let s = String.trim(read_line()) in
    let len = String.length s in
    if len >= 2 && (String.sub s (len - 2) 2) = ";;"
    then (String.sub s 0 (len - 2))
    else s ^ "\n" ^ readInput()
  in
  let rec loop() =
    print_string "D> ";
    let s = readInput() in
    let v = run s in
    printf "val: %s\n" (valToString v);
    loop()
  in loop()
```