3.5 ML^4 — 再帰的関数定義の導入

多くのプログラミング言語では，変数を宣言するときに，その定義にその変数自身を参照するという，**再帰的定義** (*recursive definition*) が許されている．ML^4 では，このような再帰的定義の機能を導入する．ただし，単純化のため再帰的定義の対象を関数に限定する．

まず，再帰的定義のための構文 let rec 式・宣言を，以下の文法で導入する．

$$
\begin{aligned}
\langle\ \text{プログラム}\ \rangle\ &::=\ \ldots \mid \mathsf{let\ rec}\ \langle\ \text{識別子}_1\ \rangle = \mathsf{fun}\ \langle\ \text{識別子}_2\ \rangle \to \langle\ \text{式}\ \rangle;; \\
\langle\ \text{式}\ \rangle\ &::=\ \ldots \\
&\quad \mid\ \ \mathsf{let\ rec}\ \langle\ \text{識別子}_1\ \rangle = \mathsf{fun}\ \langle\ \text{識別子}_2\ \rangle \to \langle\ \text{式}_1\ \rangle\ \mathsf{in}\ \langle\ \text{式}_2\ \rangle
\end{aligned}
$$

この構文の基本的なセマンティクスは let 式・宣言と似ていて，環境を宣言にしたがって拡張したもとで本体式を評価するものである．ただし，環境を拡張する際に，再帰的な定義を処理する工夫が必要になる．例で説明しよう．

```
let rec fact = fun n -¿ if n = 0 then 1 else n * (fact (n + (-1))) in
fact 5
```

おなじみの階乗関数である．この式を評価する際には，まず関数 fact を関数閉包に束縛する．この関数閉包は，n を受け取って if n = 0 then 1 else n * (fact (n + (-1))) を返す関数である．この関数閉包内には，**??**節で説明したとおり，関数閉包を作る時点での環境が保存される．いまこの環境はデフォルトの大域環境 `initial_env` と同じである．この関数閉包を fact 5 で使用している．関数適用を行う際には，関数閉包内に保存されている環境を取り出し，その環境を仮引数に対する束縛で拡張した上で関数本体の評価を行う．したがって，この例では，`initial_env` を `n=5` で拡張した環境で if n = 0 then 1 else n * (fact (n + (-1))) の部分を評価することになる．数ステップ後，インタプリタは fact (n + (-1)) をこの環境で評価することになるのだが，環境内には `fact` に対する束縛が含まれていないので，エラーとなる．

問題は何だったのだろうか．let rec fact = fun n -¿ if n = 0 then 1 else n * (fact (n + (-1))) で再帰関数を定義する際に，fact に対する束縛が関数閉包内に保存される環境に入っていなかったことである．再帰関数においては，**今これから作ろうとしている関数である fact を関数本体 if n = 0 then 1 else n * (fact (n + (-1))) 内で使う可能性があるので，fact に対する束縛も閉包内の環境に含まれていなければならない**．このような circular な構造をいかにして実現するかが再帰関数を扱う上でのキモとなる．

これを実現するための方法はいくつかあるが，今回はいわゆる**バックパッチ** (*backpatching*) と呼ばれる手法を用いる．バックパッチは，最初，ダミーの環境を用意して，ともかく関数閉包を作成し，環境を拡張してしまう．そののちダミーの環境を，たった今作った関数閉包で拡張した環境に**更新**する，という手法である．

図 3.1 が，主なプログラムの変更点である．再帰関数を定義する際に，一旦ダミーの環境を作成し，関数閉包を作成した後に，その環境を更新する必要があるが，これを OCaml の参照を用いて実現している．`eval.ml` の `exval` 型の定義において，`ProcV` が保持するデータ

syntax.ml:

```
type exp =
   ...
  | LetRecExp of id * id * exp * exp

type program =
   ...
  | RecDecl of id * id * exp
```

eval.ml:

```
type exval =
   ...
  | ProcV of id * exp * dnval Environment.t ref

let rec eval_exp env = function
 ...
 | LetRecExp (id, para, exp1, exp2) ->
    (* ダミーの環境への参照を作る *)
    let dummyenv = ref Environment.empty in
    (* 関数閉包を作り，id をこの関数閉包に写像するように現在の環境 env を拡張 *)
    let newenv =
       Environment.extend id (ProcV (para, exp1, dummyenv)) env in
    (* ダミーの環境への参照に，拡張された環境を破壊的代入してバックパッチ *)
    dummyenv := newenv;
    eval_exp newenv exp2
```

図 3.1: 再帰的関数定義

が環境 dnval Environment.t ではなく，環境への参照 dnval Environment.t ref になっていることに注意されたい.（したがって，ここに明示されていない関数適用のケースにおいては，格納されている環境を使用するために，参照から環境を取り出す操作が必要になる.）eval_exp の LetRecExp を処理する部分は，まずダミーの型環境への参照 dummyenv を作った上で，この dummyenv を含む関数閉包を作成し，現在の環境 env を id からこの関数閉包への写像で拡張した環境 newenv を作り，を上で述べたバックパッチをまさに実現している.

Exercise 3.5.1 図に示した syntax.ml にしたがって，parser.mly と lexer.mll を完成させ，ML^4 インタプリタを作成し，テストせよ．(let rec 宣言も実装すること．)

Exercise 3.5.2 [⋆⋆] and を使って変数を同時にふたつ以上宣言できるように let rec 式・宣言を拡張し，相互再帰的関数をテストせよ.

3.6 ML^5 and beyond — やりこみのための演習問題

おめでとう．ここまでやれば，一応関数型言語のインタプリタと呼べるものは出来上がったと言って良い．ここからはやりこみのための演習問題をすこし挙げておく.（今のところリストとパターンマッチに関する問題しか作れていない．ごめん.）

3.6.1 リストとパターンマッチ

Exercise 3.6.1 [⋆⋆] 今までのことを応用して，空リスト []，右結合の二項演算子::，match 式を導入して，リストが扱えるように ML^4 インタプリタを拡張せよ．match 式の構文は，

$$\text{match}\ \langle \text{式}_1 \rangle\ \text{with}\ [] \to \langle \text{式}_2 \rangle \mid \langle \text{識別子}_1 \rangle :: \langle \text{識別子}_2 \rangle \to \langle \text{式}_1 \rangle$$

程度の制限されたものでよい.

Exercise 3.6.2 [⋆] リスト表記

$$[\langle \text{式}_1 \rangle; \ldots; \langle \text{式}_n \rangle]$$

をサポートせよ.

Exercise 3.6.3 [⋆] match 式のパターン部において，リストの先頭と残りを表す変数 (:: の両側) に同じものが使われていた場合にエラーを発生するように改良せよ.

Exercise 3.6.4 [⋆⋆⋆] より一般的なパターンマッチ構文を実装せよ.

Exercise 3.6.5 [⋆⋆] ここまで与えた構文規則では，OCaml とは異なり， if，let, fun, match 式などの「できるだけ右に延ばして読む」構文が二項演算子の右側に来た場合，括弧が必要になってしまう．この括弧が必要なくなるような構文規則を与えよ．例えば，

```
1 + if true then 2 else 3;;
```

などが正しいプログラムとして認められるようにせよ.

3.6.2　自由課題

Exercise 3.6.6 [] （この課題は出来に応じて星の数を決める.）OCaml 以外の世の中にあるプログラミング言語の「ミニ」なバージョンを定義し，そのインタプリタを作れ．例えば C, C++, Java, Haskell, Scheme, VHDL, 正則表現，Rust, Go, Erlang, Prolog, LiLFeS, Eiffel, Clojure, Ruby, Perl, Python, PHP, Perl, R, MATLAB, Mathematica, Maple, sh, bash, zsh, csh, tcsh, Pascal, x86 アセンブリ, SPARC アセンブリ, MIPS アセンブリなど．どのくらい「ミニ」なものを作るかはお任せするが，あまりにミニすぎたりしょぼいものであった場合には，機能拡張の要求を出すことがある．

Exercise 3.6.7 [] （この課題は出来に応じて星の数を決める.）なにかイケているプログラミング言語を設計し，そのインタプリタを作れ．技術的な有用性を追求しても，笑いを追求してもよいが，あまりにしょぼい場合（例えば「空文字列のみがプログラムとして許され，任意のプログラムが何もしないという動作を行うプログラミング言語を設計しました！」など）は機能拡張の要求をすることがある．