3.4 ML^2 — 定義の導入

ここまで，ML プログラム中で参照できる変数は `main.ml` 中の `initial_env` であらかじめ定められた変数に限られていた．ML^2 では変数宣言の機能を，let 宣言と let 式として導入する．

変数宣言と有効範囲

OCaml の `let` 式は変数の定義と，その定義の下で評価される式が対になっている．例えば，以下の OCaml プログラム

```
let x = 1 in
let y = 2 + 2 in
(x + y) * v
```

は，変数 x を式 1 の評価結果（つまり整数値 1）に，変数 y を 式 2+2 の評価結果（つまり整数値 4）に束縛した上で，式 (x + y) * v を評価する，という意味である．(変数 v は初めに定義されている環境で 5 に束縛されていたことを思い出されたい．)

通常，変数定義には，定義が有効な場所・期間としての**有効範囲・スコープ** (*scope*) という概念が定まる．定義された変数を，そのスコープの外で参照することはできない．上の let 式中で，変数 x，y のスコープは式 (x+y)*v である．

一般に，ML^2 の let 式は，

$$\texttt{let}\ \langle\ \text{識別子}\ \rangle = \langle\ \text{式}\ \rangle\ \texttt{in}$$
$$\langle\ \text{本体式}\ \rangle$$

といった形をしているが (形式的な定義は後で示す)，⟨ 識別子 ⟩ の変数の有効範囲は ⟨ 本体式 ⟩ になる (⟨ 式 ⟩ を含まないことに注意)．また，有効範囲中でのその変数の出現は，**束縛されている** (*bound*) といい，変数自身を**束縛変数** (*bound variable*) である，という．上の例で，(x + y) * v 中の x は束縛変数である．このように，プログラムの文面のみから宣言の有効範囲や束縛の関係が決定されるとき，宣言が**静的有効範囲** (*static scope, lexical scope*) を持つといったり，変数が**静的束縛** (*static binding*) されるといったりする．これに対し，実行時まで有効範囲がわからないような場合，宣言が**動的有効範囲** (*dynamic scope*) を持つといい，変数が **動的束縛** (*dynamic binding*) されるという．また，ある式に着目したときに，束縛されていない変数を**自由変数** (*free variable*) と呼ぶ．

> **束縛変数** 「束縛変数」という概念が末永は学生のころなかなか理解できなかった記憶がある．他にもそういう人がいるかもしれないので，一応ここで説明を加えておく．束縛変数とは直観的には「名前替えをしても意味[1]が変わらない変数」のことを言う．たとえ

[1] ある変数が束縛変数か否かはシンタクティックに決まるので，ここで「意味」を持ち出すのは本当は変なのだが，わかりやすさのためにこのように言うことにする．

ば，let x = 3 in x + 2 という式は let y = 3 in y + 2 という式と（プログラムとしては）同じ意味を持っている．両者とも「何らかの変数を整数 2 であると定義し，その変数に 2 を加えた値を評価結果とする」という意味になっているからである．（ここで「何らかの変数」を前者の式は x としており，後者の式は y と取っている．）このように名前の付け替えをしても（付け替えられた名前が他の変数とかぶらない限り）式の意味として変化がないときに，その名前替えをされてよい変数を束縛変数というのである．

もう少し正確な説明を違う例を用いて加えてみよう．z + w という式を考えよう．この式においては変数 z が一回，w が一回用いられている．この z や w の「使用」のことを変数 z と w の**（自由な）出現** $(\textit{(free) occurrence})$ と言う．すなわち，式 z + w は z の自由な出現と w の自由な出現を（それぞれ一つ）含んでいる．この z の出現を w の出現に置き換えて w + w とすると，式の意味が変わる．

z + w という式を let z = 3 in の下に置くと let z = 3 in z + w という式になる．この式において z の自由な出現は存在しない．let z = 3 in z + w 中の z の出現は let z = 3 in によって束縛されているからである．z の出現が束縛されていることは，この z を名前替えしても式の意味が変わらないことからも見て取れる．実際に式 let z = 3 in z + w の意味と，z を v に名前変えした let v = 3 in v + w の意味とを比べてみると，どちらも「何らかの変数の値を 3 とおいて，3 と w の値を足す」という意味であることが見て取れよう．[2]

束縛変数の概念は記号論理学にも見られる．例えば，$\exists x \in \mathbb{R}.x \leq 1$ という一階述語論理の論理式は（$\mathbb{R}$ が実数の集合であるとすれば）「1 以下の実数が存在する」ということを言っている．この論理式を $\exists y \in \mathbb{R}.y \leq 1$ と書いてもやはり「1 以下の実数が存在する」という意味になる．前者の論理式では論理式 $x \leq 1$ 中の x は $\exists x \in \mathbb{R}$ によって束縛されている．

また，多くのプログラミング言語と同様に，ML^2 では，ある変数の有効範囲の中に，同じ名前の変数が宣言された場合，内側の宣言の有効範囲では，外側の宣言を参照できない．このような場合，内側の有効範囲では，外側の宣言の**シャドウイング** $(\textit{shadowing})$ が発生しているという．例えば，

```
(* 一つ目の x の定義 *)
let x = 2 in
let y = 3 in
(* 二つ目の x の定義 *)
let x = x + y in
x * y
```

という ML^2 の式において，一つ目の x の定義の有効範囲は，内側の let 式全体（すなわち let y = 3 in let x = x + y in x * y）であるが，二つ目の x の定義によって一つ目の定義がシャドウイングされるので，式 x * y 中では一つ目の x の定義を参照することはできない．また二つ目の x の定義の右辺に現れる x + y の x は一つ目の x の定義を参照しているので，この式の値は 15 である．実は，最初の例でも x の宣言は，大域環境で束縛されている x のシャドウイングが発生しているといえる．

[2] ここで，名前替え先の変数名については他に自由に出現している変数名と被ってはならないことに少し注意が必要である．もし z を v ではなく（すでに自由に出現している）w に名前替えしてしまうと，この式 let w = 3 in w + w となり意味が変わってしまう．

3.4.1 let 宣言・式の導入

ML^2 の構文は，以下のように与えられる．

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

Expressed value, denoted value ともに以前と同じ，つまり，let による束縛の対象は，式の値である．この拡張に伴うプログラムの変更点を図 3.1 に示す．`syntax.ml` では，構文の拡張に伴うコンストラクタの追加，`parser.mly` では，具体的な構文規則 (let は結合が if と同程度に弱い) の追加，`lexer.mll` では，予約語と記号の追加を行っている．`eval_exp` の let 式を扱う部分では，最初に，束縛変数名，式をパターンマッチで取りだし，各式を評価する．その値を使って，現在の環境を拡張し，本体式を評価している．トップレベル定義の評価（`eval_decl`）では，拡張された

Exercise 3.4.1 ML^2 インタプリタを作成し，テストせよ．

Exercise 3.4.2 [$\star\star$] OCaml では，`let` 宣言の列を一度に入力することができる．この機能を実装せよ．以下は動作例である．

```
# let x = 1
let y = x + 1;;
val x = 1
val y = 2
```

Exercise 3.4.3 [$\star\star$] バッチインタプリタを作成せよ．具体的には `miniml` コマンドの引数として ファイル名をとり，そのファイルに書かれたプログラムを評価し，結果をディスプレイに出力するように変更せよ．また，コメントを無視するよう実装せよ．(オプション: ;; で区切られたプログラムの列が読み込めるようにせよ．)

Exercise 3.4.4 [$\star\star$] 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 式・宣言を拡張せよ．例えば以下のプログラム

```
let x = 100
and y = x in x+y
```

の実行結果は 200 ではなく，(x が大域環境で 10 に束縛されているので) 110 である．

syntax.ml:

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

type program =
    Exp of exp
  | Decl of id * exp
```

parser.mly:

```
%token LET IN EQ

toplevel :
     e=Expr SEMISEMI { Exp e }
   | LET x=ID EQ e=Expr SEMISEMI { Decl (x, e) }

Expr :
     e=IfExpr { e }
   | e=LetExpr { e }
   | e=LTExpr { e }

LetExpr :
     LET x=ID EQ e1=Expr IN e2=Expr { LetExp (x, e1, e2) }
```

lexer.mll:

```
let reservedWords = [
    ...
  ("in", Parser.IN);
  ("let", Parser.LET);
]

...

| "<" { Parser.LT }
| "=" { Parser.EQ }
```

eval.ml:

```
let rec eval_exp env = function
    ...
  | LetExp (id, exp1, exp2) ->
  (* 現在の環境で exp1 を評価 *)
  let value = eval_exp env exp1 in
  (* exp1 の評価結果を id の値として環境に追加して exp2 を評価 *)
  eval_exp (Environment.extend id value env) exp2

let eval_decl env = function
     Exp e -> let v = eval_exp env e in ("-", env, v)
  | Decl (id, e) ->
      let v = eval_exp env e in (id, Environment.extend id v env, v)
```

図 3.1: 局所定義