2005年度「計算機科学実験及演習4(記号処理)」
実験資料
関数型プログラミング言語とインタプリタ

五十嵐 淳

京都大学 工学部情報学科計算機科学コース

大学院情報学研究科知能情報学専攻

工学部10号館1階142号室

e-mail: igarashi@kuis.kyoto-u.ac.jp

1 実験概要

1.1 実験の目的と内容

本実験及演習の目的は，Scheme (Lisp の方言) 風関数型言語 mini Scheme のインタプリタの作成・改造を通じて，プログラミング言語に見られる様々な機能とそのバリエーションがどのように実現されているかの理解を深めることである．インタプリタは，コンパイラと同様，プログラムという記号データを処理するプログラムであるが，実現がコンパイラに比べ簡単で，新しい機能を追加・テストするために適当である．

また，二次的な目的としては，プログラムを対象とする議論における基本的な用語を理解・習得することも挙げられる．

実験では，まず，前半 (4 週程度) で，インタプリタを実装するための言語として，関数型言語 Objective Caml の演習を，実験時に配布する Objective Caml 入門テキストを使って行う．ここでの Objective Caml の習得を通じ，関数型プログラミングに慣れ親しむことによって，作成するインタプリタの対象言語 (mini Scheme) の動作に関する直感を得る．そののち，インタプリタの作成を行う．インタプリタの作成は小さい言語から始め，徐々に機能を付け加え大きい言語へ拡張してゆく．

実験スケジュールは以下を予定している．解説は主に木曜日に行う予定である．

第 1 週	Objective Caml テキスト第 2・3 章の解説・演習
第 2 週	同第 4 章の解説・演習
第 3 週	同第 5 章の解説・演習
第 4 週	同第 6・7 章 ($+\alpha$) の解説・演習
第 5 週〜	インタプリタ作成実験

1.2 成績評価

前半の Objective Caml 演習 (50 点満点) はテキスト中の練習問題を解きレポートにする．後半のインタプリタ作成実験 (50 点満点) も，本指導書の練習問題を解きレポートにする．

必修問題とマーク (または授業中にアナウンス) したものについては，レポートの提出がない場合，大幅に減点する．必修問題が満足な出来であれば，よい成績が期待できる．その他の問題も加点の対象であるので，できるだけ解いてほしい．

レポートは，プログラムを書く場合は，なぜそのようなプログラムに至ったかの説明を加えること．これがなければ，ほとんど評価されないので注意すること．(きれいな解答を思いつけばつくほど，短く，似たようなプログラムになる傾向がある．) また，インタプリタ作成においては，ソースコードを与えた上で,「インタプリタをテストせよ」という課題があるが，この場合，テストに用いるプログラムも評価対象である．導入された機能をきちんとテストするにはどうすればいいかを考えること．

1.3 資料，参考書，マニュアル

授業の web ページの URL は `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/class/isle4/` である．ここで本稿の完全版や「Objective Caml 入門」のテキストが利用可能になる．また，Objective Caml のマニュアル [4] (英語) も `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/class/isle4/ocamlman/` よりオンライン利用が可能なようにしてある．その他，`http://caml.inria.fr/` から，FAQ などの文書，Objective Caml を使ったソフトウェアなどが利用できる．Objective Caml は，Caml という言語を拡張して，オブジェクト指向プログラミングの機能などを加えたものであるが，本実験ではオブジェクト指向プログラミング機能をとくに必要としないため，本来の Caml の教科書 [1] も参考になる．また，フランス語の Objective Caml の本が O'Reilly から出版されているが，現在，英訳プロジェクトが進行中で，オンラインで `http://caml.inria.fr/oreilly-book/` より利用可能になっている．

後半のインタプリタ作成に関しては，[3, 第 3 章] を参考にしている．(この本では，Objective Caml ではなく Scheme 自身で Scheme インタプリタを作成している．) Scheme 言語に関しては，特に学習する必要はないが，[6, 2] などが参考になる．

2 Objective Caml 入門

別に配布するテキストを参照のこと．以下は，テキストからの Objective Caml 言語に関する記述の一部抜粋である．

2.1 Objective Caml とは

プログラミング言語 ML は，元々は計算機による証明支援系から発展してきた言語[1]で，関数型プログラミングと呼ばれるプログラミングスタイルをサポートしている．ML は核となる部分が小さくシンプルであるため，プログラミング初心者向けの教育用に適した言語であると同時に，大規模なアプリケーション開発のためのサポート (モジュールシステム・ライブラリ) が充実している．ML の核言語は型付き λ 計算と呼ばれる，形式的な計算モデルに基づいている．このことは，言語仕様を形式的に (数学的な厳密な概念を用いて) 定義し，その性質を厳密に「証明」することを可能にしている．実際，Standard ML という言語仕様 [5] においては，(コンパイラの受理する) 正しいプログラムは決して未定義の動作をおこさない，といった性質が証明されている．

この実験及演習で学ぶのは ML の方言である Objective Caml という言語である．Objective Caml は INRIA というフランスの国立の計算機科学の研究所でデザイン・開発された言語で，Standard ML とは文法的には違った言語であるが，ほとんどの機能は共有している．また，OCaml では Standard ML には見られない，独自の拡張が多く施されており，関数型プ

[1] 数学的証明を形式的記述するための対象言語 (object language) に対して，証明戦略の記述用の言語であるメタ言語 (Meta Language) の頭文字をとった．

ログラミングだけでなく，オブジェクト指向プログラミングもサポートされている．またコンパイラも効率的なコードを生成する優れたものが開発されている．

2.2 ML (Objective Caml) の特徴

ML (Objective Caml) の特徴としては，以下のようなものが挙げられる．

- **高階関数**(*higher-order function*) の導入により，関数を他の関数への引数として渡したり，計算結果として受け取ることができる，
- **パターンマッチング**(*pattern matching*) による，より宣言的なプログラミングのサポート
- **静的型システム**(*static type system*) による安全性の保証
- **多相型システム**(*polymorphic type system*) により，プログラム中の一つの式に，複数の型を割り当てることが可能になり，コード再利用性が高まる．
- **型推論**(*type inference*) により，煩雑な型宣言を省略できる．
- 強力な**モジュール・システム**(*module system*) により，大規模プログラミングに必要な**分割コンパイル**(*separate compilation*) や，**抽象データ型**(*abstract data type*) による，プログラム部品間の情報隠蔽・言語に新たな基本データ型を加えるようなプログラミングが可能になる．また，**ファンクタ**(*functor*) と呼ばれる，**パラメータを持つモジュール**(*parameterized module*) により，再利用性の高い，大規模プログラミングがサポートされる．
- **ごみ集め**(*garbage collection*) により，自動メモリ管理が行われるため，プログラマは C 言語の `malloc`/`free` などを使ったメモリ管理のように頭を悩ませる必要がない．
- 対話的にプログラム開発ができるインタラクティブ・コンパイラと，ソースコードを一括して実行形式に変換するバッチ・コンパイラが利用できる．
- 関数型プログラミングを核とした言語に加えてのオブジェクト指向プログラミングの導入．(Objective Caml 独自の特徴)

3 mini Schemeインタプリタの作成

3.1 インタプリタとは

インタプリタ(*interpreter*) は，文字列を受け取って，それを特定のプログラミング言語のプログラムとして解釈して，実行結果を計算する計算機プログラムである．よってインタプリタは，解釈するプログラミング言語の**シンタックス**(*syntax*)，つまり，どのような文字列

がプログラムをなすか，と，**セマンティクス**(*semantics*)，つまり，プログラムがどのように実行されるか，のふたつを間接的に定義しているといえる．コンパイラもまた，あるプログラミング言語の構文と意味を規定しているが，入力プログラムから，その実行結果を計算するかわりに，別の (多くの場合，アセンブリ言語などより低級な) プログラミング言語に翻訳した結果を出力とするプログラムである．その意味では，インタプリタとコンパイラではセマンティクスの与え方が違っているといえる．

さて，インタプリタ自体もプログラムであるから，なんらかのプログラミング言語で書かれている．このとき,「インタプリタ自体が書かれているプログラミング言語」を**定義する言語**(*defining language*) といい,「インタプリタが入力として受け取るプログラミング言語」を**定義される言語**(*defined language*) という[2]． 本実験では,

$$\begin{aligned} \text{定義する言語} &= \text{Objective Caml} \\ \text{定義される言語} &= \text{mini Scheme} \end{aligned}$$

として進めていく．Objective Caml プログラムの記述にはタイプライタ体 (abcde) を，mini Scheme プログラムにはサン・セリフ体 (abcde) を用いて区別する．

典型的なインタプリタは，字句解析・構文解析・解釈部から構成される．字句解析・構文解析はコンパイラと同様に，文字列からプログラムの抽象構文木を生成する過程で，定義される言語のシンタックスを規定している．解釈部分は，セマンティクスを定義していて，抽象構文木を入力としてプログラムの実行結果を計算する部分で，インタプリタの核となる．

3.2 プログラムファイルの構成・コンパイル方法

本実験で作成するインタプリタプログラムは，以下の 5 つのファイルから構成される．

`syntax.ml` このファイルでは，抽象構文木のデータ構造 (つまり抽象構文) を定義している．抽象構文木は構文解析の出力であり，解釈部の入力なので，インタプリタの全ての部分が，この定義に (直接/間接的に) 依存する．

`parser.mly` C 言語に yacc や bison といった構文解析プログラム生成ツールがあるように，Objective Caml にも ocamlyacc というツールがあり，`.mly` という拡張子のファイルに記述された文法定義から，構文解析プログラムを生成する．文法定義の仕方は yacc と似ている．

`lexer.mll` C 言語に lex, flex といった字句解析プログラム生成ツールがあるように，Objective Caml にも ocamllex というツールがあり，`.mll` という拡張子のファイルに定義されたトークンとなる文字列のパターン定義から，字句解析プログラムを生成する．パターンの定義の仕方は lex と似ている．

[2]多くの場合，定義する言語と定義される言語は異なるが，一致する場合もありうる，例えば Objective Caml で Objective Caml インタプリタを記述することも可能である．このような場合，そのインタプリタを特に，**メタ・サーキュラ・インタプリタ**(*meta circular interpreter*) という．

core.ml 解釈部プログラムである．構文解析部が生成した構文木から計算を行なう．

main.ml 字句解析・構文解析・解釈部を組み合わせて，インタプリタ全体を機能させる．プログラム全体の開始部分でもある．

最初に扱う mini Scheme[1] インタプリタのための 5 つのファイルに加え，インタプリタをコンパイルするための Makefile を http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/class/isle4/src/ に置いてある．これら 6 つのファイルを同じディレクトリに保存し，どれかひとつの .ml ファイルを Emacs に読み込む．そのバッファで C-c C-c とすると，コンパイルのコマンドを聞かれるので， make depend とする．この作業は初回のみ (正確にはファイルが増えた時もしくは make clean を行なった後) 行えばよい． 次に，C-c C-c make -k とする． すると，ソースファイルのあるディレクトリに scm という実行形式ファイルが生成される．(M-x shell でシェルモードに 入って) scm を起動すると => というプロンプトが現れるので， mini Scheme プログラムを打つと結果が表示される．

```
> scm
⇒ x        ;; 起動時に大域変数 x の値は 10 になっている．
10
⇒ (+ x 3)
13
```

ソースを変更したあとは，C-c C-c make -k でコンパイルすることになる．コンパイル時にエラーが発生した場合は M-x next-error とすることで，エラーの発生した場所にカーソルが移動する．

実行可能ファイルとなった Objective Caml プログラムをデバッグするには ocamldebug を使用する方法 (Objective Caml マニュアル 16 章参照) と，インタラクティブコンパイラ ocaml にプログラムを構成する各モジュールをロードして，テストする方法がある．ocaml を起動する際に

```
ocaml -I⟨ モジュールのあるディレクトリのパス ⟩ foo.cmo bar.cmo ...
```

のようにオブジェクトファイルを指定すると，Foo, Bar というモジュールが利用できるようになるので，トップレベルでテストすることが可能になる．

3.3 mini Scheme[1] インタプリタ — プリミティブ演算と環境を使った変数参照

まず，非常に単純な言語として，整数，変数参照と加減乗演算のみ (新しい変数の宣言すらできない!) を持つ言語 mini Scheme[1] から始める．

最初に，mini Scheme[1] の (具象) 文法を以下のように与える．

⟨ プログラム ⟩ ::= ⟨ 式 ⟩

```
(* abstract syntax *)
type id = string

type prim = Plus | Minus | Mul | Add1 | Sub1

type exp =
    ILit of int             (* Integer LITeral *)
  | Var of id               (* VARiable reference *)
  | Prim of prim * exp list (* PRIMitive application *)

type program = Prog of exp
```

図 1: mini Scheme[1] インタプリタ: syntax.ml

$$
\begin{aligned}
\langle \text{式} \rangle &::= \langle (0 \text{ 以上の}) \text{ 整数} \rangle \\
&\mid \langle \text{識別子} \rangle \\
&\mid (\langle \text{プリミティブ} \rangle \langle \text{式}_1 \rangle \ldots \langle \text{式}_n \rangle) \\
\langle \text{プリミティブ} \rangle &::= + \mid - \mid * \mid \mathsf{add1} \mid \mathsf{sub1}
\end{aligned}
$$

プログラムは，ひとつの式からなり，式は，(0 以上の) 整数，識別子による変数参照，またはプリミティブ適用式のいずれかである．識別子は，英小文字で始まり，数字・英小文字・「'(アポストロフィ)」を並べた文字列である．例えば，以下の文字列はいずれも mini Scheme[1] プログラムである．

```
3
x
(+ 3 x')
(add1 (+ 3 x1))
```

それでは，構文に関する部分から順に，5 つのファイルを見ていく．

3.3.1 syntax.ml: 抽象構文のためのデータ型

上の文法に対する，抽象構文木のためのデータ型は図 1 のように宣言される．id は変数の識別情報を示すための型で，ここでは変数の名前を表す文字列としている．(より現実的なインタプリタ・コンパイラでは，変数の型などの情報も加わることが多い．) prim，exp，program 型に関しては上の文法構造をそのまま写した形の宣言になっていることがわかるだろう．

3.3.2 parser.mly, lexer.mll: 字句解析と構文解析

ocamlyacc は，yacc と同様に，LALR(1) の文法を定義したファイルから構文解析プログラムを生成するツールである．ここでは，LALR(1) 文法や構文解析アルゴリズムなどに関

しての説明などは割愛し (コンパイラの教科書などを参照のこと)，文法定義ファイルの説明を parser.mly を具体例として行う．

文法定義ファイルは一般に，以下のように 4 つの部分から構成される．

```
%{
  〈ヘッダ〉
%}
  〈宣言〉
%%
  〈文法規則〉
%%
  〈トレイラ〉
```

〈ヘッダ〉, 〈トレイラ〉は Objective Caml のプログラムを書く部分で，ocamlyacc が生成する parser.ml の，それぞれ先頭・末尾にそのまま埋め込まれる．〈宣言〉はトークン (終端記号) や，開始記号，優先度などの宣言を行う．parser.mly では演習を通して，開始記号とトークンの宣言のみを使用する．〈文法規則〉には文法記述と還元時のアクションを記述する．ヘッダ・トレイラでは，コメントは Objective Caml と同様 (* ... *) であり，宣言・文法規則部分では C 言語と同様な記法 /* ... */ で記述する．

それでは parser.mly を見てみよう (図 2)．この文法定義ファイルではトレイラは空になっていて，その前の %% は省略されている．

- ヘッダにある open 宣言は，文法規則宣言部で syntax.ml 中で宣言されているコンストラクタ・型の名前をモジュール名無しで使えるようにしている．(これがないと，Syntax.Var などの長い名前で参照しなくてはならない．)

- %token 〈トークン名〉 ... は，属性を持たないトークンの宣言である．ここでは括弧 “(” と “)” に対応するトークン LPAREN, RPAREN と，プリミティブ (+, −, ∗, add1, sub1) に対応するトークン PLUS, MINUS, MUL, ADD1, SUB1 が宣言されている．(図 1 に現れる構文木のコンストラクタ Plus などとの区別に注意すること．) この宣言で宣言されたトークン名は ocamlyacc の出力する parser.ml 中で，token 型の (引数なし) コンストラクタになる．通常，字句解析プログラムはこの型の値を出力する．

- %token <〈型〉> 〈トークン名〉 ... は，属性つきのトークン宣言である．数値のためのトークン INTV (属性はその数値情報なので int 型) と変数のための ID (属性は変数名を表す Syntax.id 型[3]) を宣言している．この宣言で宣言されたトークン名は parser.ml 中で，〈型〉を引数とする token 型のコンストラクタになる．

- %start 〈開始記号名〉 ... で (一つ以上の) 開始記号の名前を指定する．ocamlyacc が生成する，parser.ml ファイルでは，同名の関数が構文解析関数として宣言される．ここでは toplevel という名前を宣言しているので，後述する main.ml では Parser.toplevel という関数を使用して構文解析をしている．開始記号の名前は，次の %type 宣言でも宣言されていなくてはならない．

[3]ヘッダ部の open 宣言はトークン宣言部分では有効ではないので，Syntax. をつけることが必要である．

```
%{
open Syntax
%}

%token LPAREN RPAREN
%token PLUS MINUS MUL ADD1 SUB1

%token <int> INTV
%token <Syntax.id> ID

%start toplevel
%type <Syntax.program> toplevel
%%

toplevel :
    Exp { Prog $1 }

Exp :
    INTV { ILit $1 }
  | ID { Var $1 }
  | LPAREN PrimOp Arglist RPAREN { Prim ($2, $3) }

PrimOp :
    PLUS { Plus }
  | MINUS { Minus }
  | MUL { Mul }
  | ADD1 { Add1 }
  | SUB1 { Sub1 }

Arglist :
 /* empty */ { [] }
  | Exp Arglist { $1 :: $2 }
```

図 2: mini Scheme[1] インタプリタ: parser.mly

- %type <⟨型⟩> ⟨名前⟩ ... 名前の属性を指定する宣言である，toplevel はひとつのプログラムの抽象構文木を表すので属性は Syntax.program 型となっている．

- 文法規則は，

  ⟨非終端記号名⟩ :
  　⟨記号名 $_{11}$⟩ ... ⟨記号名 $_{1n_1}$⟩ { ⟨還元時アクション $_1$⟩ }
  | ⟨記号名 $_{21}$⟩ ... ⟨記号名 $_{2n_2}$⟩ { ⟨還元時アクション $_2$⟩ }
  ...

  のように記述する．⟨還元時アクション⟩には Objective Caml の式を記述する．$\$i$ で，i 番目の記号の属性を参照することができる．式全体の評価結果がこの非終端記号の属性となるので，式の型は全て一致している必要がある．例えば Exp は Syntax.exp 型の値 (つまり mini Scheme[1] の式の抽象構文木) を属性として持ち，各アクションでは，その生成規則に対応する抽象構文木を生成するような式が書かれている．

さて，この構文解析器への入力となるトークン列を生成するのが字句解析器である．ocamllex は lex と同様に正則表現を使った文字列パターンを記述したファイルから字句解析プログラムを生成する．.mll ファイルは，

```
{ ⟨ヘッダ⟩ }

let ⟨名前⟩ = ⟨正則表現⟩
...

rule ⟨エントリポイント⟩ =
  parse ⟨正則表現⟩ { ⟨アクション⟩ }
    |   ⟨正則表現⟩ { ⟨アクション⟩ }
    |   ...
and ⟨エントリポイント⟩ =
  parse ...
and ...
{ ⟨トレイラ⟩ }
```

という構成になっている．ヘッダ・トレイラ部には，Objective Caml のプログラムを書くことができ，ocamllex が生成する lexer.ml ファイルの先頭・末尾に埋め 込まれる．次の let を使った定義部は，よく使う正則表現に名前をつけるための部分で，lexer.mll では何も定義されていない．続く部分がエントリポイント，つまり字句解析の規則の定義で，同名の関数が ocamllex によって生成される．規則としては正則表現とそれにマッチした際のアクションを (Objective Caml 式で) 記述する．アクションは，基本的には (parser.mly で宣言された) トークン (Parser.token 型) を返すような式を記述する．また，字句解析に使用する文字列バッファが lexbuf という名前で使えるが，通常は以下の使用法でしか使われない．

- Lexing.lexeme lexbuf で，正則表現にマッチした文字列を取り出す．

- Lexing.lexeme_char lexbuf n で，マッチした文字列の n 番目の文字を取り出す．

```
{
let reservedWords = [
  (* Keywords *)
  ("+", Parser.PLUS);
  ("-", Parser.MINUS);
  ("*", Parser.MUL);
  ("add1", Parser.ADD1);
  ("sub1", Parser.SUB1);
]
}

rule main = parse
  (* ignore spacing and newline characters *)
  [' ' '\009' '\012' '\n']+     { main lexbuf }

| ['0'-'9']+
    { Parser.INTV (int_of_string (Lexing.lexeme lexbuf)) }

| "(" { Parser.LPAREN }
| ")" { Parser.RPAREN }

| ['a'-'z'] ['a'-'z' '_' '0'-'9' '\'']*
| ['+' '-' '*']
    { let id = Lexing.lexeme lexbuf in
      try
        List.assoc id reservedWords
      with
      _ -> Parser.ID id
    }
| eof { exit 0 }
```

図 3: mini Scheme[1] インタプリタ: `lexer.mll`

- `Lexing.lexeme_start lexbuf` で，マッチした文字列の先頭が入力文字列全体でどこに位置するかを返す．末尾の位置は `Lexing.lexeme_end lexbuf` で知ることができる．
- ⟨エントリポイント⟩ `lexbuf` で，⟨エントリポイント⟩規則を呼び出す．

それでは，具体例 `lexer.mll` を使って説明を行う．ヘッダ部では，予約語の文字列と，それに対応するトークンの連想リストである，`reservedWords` を定義している．後でみるように，`List.assoc` 関数を使って，文字列からトークンを取り出すことができる．

エントリポイント定義部分では，`main` という (唯一の) エントリポイントが定義されている．最初の正則表現は空白やタブなど文字の列にマッチする．これらは mini Scheme では区切り文字として無視し，次のトークンを求めるために `main lexbuf` を呼び出している．次は，数字の並びにマッチし，`int_of_string` を使ってマッチした文字列を `int` 型に直して，トークン `INTV` (属性は `int` 型) を返す．続くふたつは開き・閉じ括弧である．次は識別子の

ための正則表現で，英小文字で始まる名前か，演算記号にマッチする．アクション部では，マッチした文字列が予約語に含まれていれば，予約語のトークンを，そうでなければ (例外が発生した場合は) `ID` トークンを返す．最後の `eof` はファイルの末尾にマッチする特殊なパターンである．ファイルの最後に到達したら `exit` するようにしている．

なお，この部分は，今後もあまり変更が必要がないので，正則表現を記述するための表現についてはあまり触れていない．興味のあるものは lex を解説した本，Objective Caml マニュアルを参照すること．

3.3.3 `core.ml`: 解釈部

Expressed value と Denoted value さて，本節冒頭でも述べたように，解釈部は，定義される言語のセマンティクスを定めている．プログラミング言語のセマンティクスを定めるに当たって重要なことは，どんな類いの値をプログラムが操作できるかを定義することである．この時，**式の値**(*expressed value*) の集合と**変数が指示する値**(*denoted value*) の集合を区別する[4]．mini Scheme[1] では，このふたつは一致するが，これらが異なる言語も珍しくない．実際，mini Scheme[6] で，言語に変数への代入を導入することで，両者に違いが現れる．

$$\begin{aligned} \text{Expressed Value} &= \text{整数}\,(\ldots,-2,-1,0,1,2,3,\ldots) \\ \text{Denoted Value} &= \text{整数} \end{aligned}$$

であるとする．

このための型宣言を以下に示す．

```
(* Expressed values *)
type exval =
    IntV of int

(* Denoted values *)
and dnval = exval
```

`exval` 型はコンストラクタがひとつのヴァリアント型で表現しているが，これは，将来，式の値の集合に整数以外のものが入ってきたときの，コードの変更を容易にするためである．

環境 もっとも簡単な解釈部の構成法のひとつは，抽象構文木と，変数・denoted value 間の束縛状態の組から，実行結果を計算する方式である．この，変数の束縛状態を表現するデータ構造を**環境**(*environment*) といい，この方式で実装されたインタプリタ (解釈部) を**環境渡しインタプリタ**(*environment passing interpreter*) ということがある．

まずは，環境の型を `env` として，環境を操作する関数の型と例外を示す．

```
val empty_env : unit -> env
val extend_env : Syntax.id list -> dnval list -> env -> env
val apply_env : Syntax.id -> env -> dnval
exception UnboundVar of string
```

[4] この区別はコンパイラの教科書で見られる**左辺値**(*L-value*)，**右辺値**(*R-value*) と関連する

```
type env =
    EmptyEnv
  | ExtendEnv of id list * dnval array * env

exception UnboundVar of string

let empty_env () = EmptyEnv

let extend_env ids dnvals env =
  (* assumes List.length syms = List.length dnvals *)
  ExtendEnv (ids, Array.of_list dnvals, env)

let rec list_pos n = function
    [] -> raise Not_found
  | m :: rest -> if n = m then 0 else succ (list_pos n rest)

let rec apply_env id = function
    EmptyEnv -> raise (UnboundVar id)
  | ExtendEnv (ids, dnvals, rest) ->
      (try dnvals.(list_pos id ids) with Not_found -> apply_env id rest)
```

図 4: mini Scheme[1] インタプリタ: 環境の実装 (core.ml)

最初の関数 empty_env は，empty_env () とすると，何の変数も束縛されていない，空の環境を生成する．次の extend_env は，環境に新しい束縛をいくつか同時に付け加えるための関数で，extend_env ids dnvals env で，環 境 env に対して，変数名のリスト ids の i 番目の要素である変数を， denoted value のリスト dnvals の i 番目の要素に束縛したような新しい環境を表す．最後の apply_env 関数は，環境から変数が束縛された値を取り出すもので，apply_env id env で，環境 env の中を，新しく加わった束縛から順に変数 id を探し，束縛されている値を返す．変数が環境中に無い場合は，例外 UnboundVar が発行される．

この関数群を実装したものが図 4 である．環境のデータ表現は，ヴァリアント型を使っており，コンストラクタ EmptyEnv が空の環境を，ExtendEnv (ids, varray, env) が，環境 env に，変数名のリスト ids と denoted value の配列 varray で表現される束縛を付け加えたような環境を表現している．apply_env での，補助関数 list_pos と，例外の使用法に注意されたい．

また，プログラム実行開始時の環境 (大域環境) を i，v，x がそれぞれ 1，5，10 に束縛されたような環境として

```
let global_env =
  extend_env
    ["i"; "v"; "x"]
    (List.map (fun i -> IntV i) [1; 5; 10])
    (empty_env())
```

と定義する．この大域環境は主に変数参照のテスト用で，(空でなければ) 何でもよい．

```
let apply_prim p args =
  match p, args with
  | (Plus, [IntV i; IntV j]) -> IntV (i + j)
  | (Plus, _) -> failwith "Arity mismatch: +"
  | (Minus, [IntV i; IntV j]) -> IntV (i - j)
  | (Minus, _) -> failwith "Arity mismatch: -"
  | (Mul, [IntV i; IntV j]) -> IntV (i * j)
  | (Mul, _) -> failwith "Arity mismatch: *"
  | (Add1, [IntV i]) -> IntV (i + 1)
  | (Add1, _) -> failwith "Arity mismatch: add1"
  | (Sub1, [IntV i]) -> IntV (i - 1)
  | (Sub1, _) -> failwith "Arity mismatch: sub1"

let rec eval_exp env = function
    ILit i -> IntV i
  | Var sym -> apply_env sym env
  | Prim (p, es) ->
      let args = eval_rands env es in
      apply_prim p args

and eval_rands env = function
    [] -> []
  | e :: rest -> eval_exp env e :: eval_rands env rest

let eval_program (Prog e) = eval_exp global_env e
```

図 5: mini Scheme[1] インタプリタ: 評価部の実装 (core.ml)

解釈部の主要部分　以上の準備をすると，残りは，プリミティブ適用式を実行する部分と式を評価する部分である．前者を apply_prim, 後者を eval_exp という関数として図 5 のように定義する．eval_exp では，リテラル数値 (ILit) はそのまま値に，変数は apply_env を使って値を取りだし，プリミティブ適用式は，引数となる式 (オペランド) をそれぞれ評価し (eval_rands)，apply_prim を呼んでいる．apply_prim は与えられたプリミティブにしたがって，対応する Objective Caml の演算をしている．引数の数のチェックはパターンマッチで行っている．

3.3.4 main.ml

メインプログラム main.ml を図 6 に示す．関数 run で，字句解析・構文解析・解釈部の結合を行っている．lexer.mll で宣言された規則の名前 main が関数 Lexer.main に，parser.mly (の %start) で宣言された非終端記号の名前 toplevel が関数 Parser.toplevel に対応している．Parser.toplevel は第一引数として構文解析器から呼び出す字句解析器を，第二引数として読み込みバッファを表す Lexing.lexbuf 型の値 (ここでは標準入力から Lexing.from_channel を使って作られている) をとっている．関数 read_eval_print では，

```
let run () =
  Core.eval_program
    (Parser.toplevel Lexer.main (Lexing.from_channel stdin))

let rec read_eval_print () =
  print_string "=> ";
  flush stdout;
  Core.pp (run ());
  print_newline ();
  read_eval_print ()

let _ = read_eval_print ()
```

図 6: mini Scheme[1] インタプリタ: `main.ml`

まず，プロンプトを出力し，`run` の呼び出しによるプログラムの入力と評価を行い，その結果を `core.ml` に定義された `pp` という関数を使って出力し，自分自身にループしている．最後の `let _ = ...` (右辺の評価結果は使わない (実際には実行は終了しない) ので左辺はワイルドカードパターンにしている) で，`read_eval_print` を呼出し，インタプリタの実行を開始する．

Exercise 3.1 [**必修課題**] mini Scheme[1] インタプリタのプログラムをコンパイル・実行し，インタプリタの動作を確かめよ．大域環境として i, v, x の値のみが定義されているが，ii が 2，iii が 3，iv が 4 となるようにプログラムを変更して，動作を確かめよ．例えば，

(- (sub1 (* iii (+ ii v))) iv)

などを試してみよ．

Exercise 3.2 [⋆] 本来の Scheme 言語では + や ∗ は以下のように，任意個の引数を受け取れる．

⇒ (+ 2 3 4)
9
⇒ (* 3 4 5)
60

+ や ∗ がこのように動作するように変更せよ．

Exercise 3.3 [⋆] このインタプリタは文法にあわない入力を与えたり，束縛されていない変数を参照しようとすると，プログラムの実行が終了してしまう．このような入力を与えた場合，適宜メッセージを出力して，インタプリタプロンプトに戻るように改造せよ．

Exercise 3.4 [⋆] バッチインタプリタを作成せよ．具体的には `scm` コマンドの引数としてファイル名をとり，そのファイルの内容を mini Scheme プログラムとして解釈し，結果をディスプレイに出力するように変更せよ．また，Lisp 系言語では ; 以降行末まではコメントとして扱われる．この機能を実装せよ．

3.4 mini Scheme^2 — 条件分岐の導入

さて，次に条件分岐の機能を追加した mini Scheme^2 を定義する．そのために，条件判定式と比較プリミティブ $=, <$ を追加する．条件判定式は (if ⟨式$_1$⟩ ⟨式$_2$⟩ ⟨式$_3$⟩) で表され，⟨式$_1$⟩ の値が 0 でなければ⟨式$_2$⟩ を評価し，その値を全体の式の値とし，0 であれば⟨式$_3$⟩ の値を全体の値とするようなものである．(= ⟨式$_1$⟩ ⟨式$_2$⟩) は両式の値が等しければ，1 を，そうでなければ 0 を返すプリミティブである．同様に，(< ⟨式$_1$⟩ ⟨式$_2$⟩) は⟨式$_1$⟩ の値が⟨式$_2$⟩ の値より小さければ 1，そうでなければ 0 を返す．

以下に，mini Scheme^2 プログラムの実行例を示す．

```
⇒ (< 5 4)
0
⇒ (if (< (+ 3 5) (- 9 18)) v x)
10
⇒ (+ (= 3 3) 5)
6
```

全体としての文法は，

$$
\begin{aligned}
\langle \text{プログラム} \rangle &::= \langle \text{式} \rangle \\
\langle \text{式} \rangle &::= \ldots \\
&\quad | \quad (\text{if } \langle \text{式}_1 \rangle \langle \text{式}_2 \rangle \langle \text{式}_3 \rangle) \\
\langle \text{プリミティブ} \rangle &::= \ldots \mid = \mid <
\end{aligned}
$$

となる．また，セマンティクスに関する定義は，真偽値も整数で表現しているので，expressed value, denoted value ともにそのままである．

3.4.1 プリミティブの追加

まずは，プリミティブの追加の方法を見る．まず，構文的な側面に関しては，このインタプリタではプリミティブをキーワードとし，文法においても特別扱いしているため，抽象構文の定義，構文解析のトークン宣言，字句解析のキーワードリストの変更が必要である．また，当初の字句解析規則では $=, <$ にマッチする規則がなかったので規則の変更も必要である．

また，`core.ml` は，プリミティブを追加したのだから，`Syntax.prim` 型に関わる部分，つまり，`apply_prim` を変更すればよい．

3.4.2 if 式の追加

if 式は，if をキーワードとして追加し，式の構文に if 式の追加が行われる．解釈部に関しては，`eval_exp` で if 式を扱う節を追加することになる．テストする式の値は Objective Caml 上では `int` 型ではなく `IntV` コンストラクタが付加された値として表されているので，パターンマッチを使ってコンストラクタを外さなければならない．

syntax.ml:

```
    type prim = Plus | Minus | Mul | Add1 | Sub1 | Eq | Lt
```

parser.mly:

```
    ...
    %token EQ LT
    ...
    PrimOp:
      ...
      | EQ { Eq }
      | LT { Lt }
```

lexer.mll:

```
    let reservedWords = [
      ...
      ("=", Parser.EQ);
      ("<", Parser.LT);
    ]

    rule main = parse
      ...
    | ['a'-'z'] ['a'-'z' '_' '0'-'9' '']*
    | ['+' '-' '*' '=' '<']
      ...
```

core.ml:

```
    let apply_prim p args =
      match p, args with
        ...
      | (Eq, [IntV i; IntV j]) -> if i = j then IntV 1 else IntV 0
      | (Eq, _) -> failwith "Arity mismatch: ="
      | (Lt, [IntV i; IntV j]) -> if i < j then IntV 1 else IntV 0
      | (Lt, _) -> failwith "Arity mismatch: <"
```

図 7: 比較プリミティブの追加

syntax.ml:

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

parser.mly:

```
...
%token IF
...
Exp :
  ...
  | LPAREN IF Exp Exp Exp RPAREN { If ($3, $4, $5) }
```

lexer.mll:

```
let reservedWords = [
  ...
  ("if", Parser.IF);
]
```

core.ml

```
let rec eval_exp env = function
  ...
  | If (e1, e2, e3) ->
      let IntV test = eval_exp env e1 in
      if test <> 0 then eval_exp env e2
      else eval_exp env e3
```

図 8: if 式の追加

Exercise 3.5 [**必修課題**] mini Scheme[2] インタプリタを作成し，テストせよ．

Exercise 3.6 [$\star$] 論理値演算プリミティブ and, or, not を追加せよ．

Exercise 3.7 [$\star\star$] 真偽値を整数で代用するのではなく，新たな mini Scheme の値として導入せよ．つまり，

$$\begin{aligned} \text{Expressed Value} &= \text{整数}\,(\ldots,-2,-1,0,1,2,3,\ldots) + \text{真偽値}\,(\mathsf{true},\mathsf{false}) \\ \text{Denoted Value} &= \text{整数} + \text{真偽値} \end{aligned}$$

とする．ヒントとして，`core.ml` の変更点を示す．

```
type exval =
    IntV of int
  | BoolV of bool
```

また，true, false はキーワードもしくは，大域環境に束縛された変数として導入する．以下は，実行例である．

```
⇒ true
true
⇒ (> 3 5)
false
⇒ (if (> 5 (+ 2 4)) x i)
1
```

(ヒント: この問題は，mini Scheme[4] までやってから戻るとやりやすい)

3.5 mini Scheme[3] — 局所定義の導入

ここまで，mini Scheme 中で参照できる変数は `core.ml` 中の `global_env` であらかじめ定められた変数に限られていた．mini Scheme[3] では変数宣言の機能を導入する．

3.5.1 変数宣言と有効範囲

ここで導入する変数宣言のための構文は let 式と呼ばれ，宣言のみを独立して導入できるものではなく，宣言と，その宣言の下で評価される式が対になるものである．例えば，以下の mini Scheme[3] プログラム

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

は，x は 1，y は (+ 2 2) の値 (つまり 4) であると宣言した下で，式 (* (+ x y) v) を評価する，という意味である．

通常，宣言には，宣言が有効な場所・期間としての**有効範囲・スコープ**$(scope)$ という概念が定まる．有効範囲をはずれたところで宣言を参照することはできない．上の let 式中で，宣言された変数 x，y のスコープは (* (+ x y) v) である．一般に，mini Scheme[3] の let 式は，

(let ((⟨ 識別子 $_1$⟩ ⟨ 式 $_1$⟩)
⋮
(⟨ 識別子 $_n$⟩ ⟨ 式 $_n$⟩))
⟨ 本体式 ⟩)

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 $_i$⟩の変数の有効範囲は ⟨ 本体式 ⟩ になる (⟨ 式 $_i$⟩ 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．

また，多くのプログラミング言語と同様に，mini Scheme 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**$(shadowing)$ が発生しているという．例えば，

```
(let ((x 2) (y 3))
  (let ((x (+ x y)))
    (* x y)))
```

において，最初の x の宣言の有効範囲は，内側の let 式全体であるが，内側にも x が宣言されているので，(* x y) 中の x は内側の宣言に束縛される．また (+ x y) の x は外側の宣言に束縛されるので，この式の値は，15 である．実は，最初の例でも x の宣言は，大域環境に束縛されている x のシャドウイングが発生しているといえる．

3.5.2 let 式の導入

mini Scheme[3] の構文は，以下のように与えられる．

⟨ 式 ⟩ ::= ...
| (let (⟨ 束縛リスト ⟩) ⟨ 本体 ⟩)
⟨ 束縛リスト ⟩ ::= (⟨ 識別子 $_1$⟩ ⟨ 式 $_1$⟩) ... (⟨ 識別子 $_n$⟩ ⟨ 式 $_n$⟩)

syntax.ml:

```
type exp =
  ...
| Let of (id * exp) list * exp
```

parser.mly:

```
%token LET

Exp :
   ...
 | LPAREN LET LPAREN Bindings RPAREN Exp RPAREN { Let ($4, $6) }

...
Bindings:
 /* empty */ { [] }
 | LPAREN ID Exp RPAREN Bindings { ($2, $3) :: $5 }
```

lexer.mll:

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

core.ml:

```
let rec eval_exp env = function
   ...
 | Let (bs, e) ->
     let ids, args = List.split bs in
     let arg_vals = eval_rands env args in
     eval_exp (extend_env ids arg_vals env) e
```

図 9: 局所定義

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．プログラムの変更点を図 9 に示す．`eval_exp` の let 式を扱う部分では，最初に，束縛変数名，式を取りだし，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．

Exercise 3.8 [**必修課題**] mini Scheme[3] インタプリタを作成し，テストせよ．

Exercise 3.9 [⋆] ここで導入した let 式は，変数を「同時に」宣言するもので，宣言されている変数をその宣言内で参照することはできなかった．以下の文法

$$
\begin{aligned}
\langle \text{式} \rangle &::= \ \ldots \\
&\quad | \quad (\text{letseq } (\langle \text{束縛リスト} \rangle)\ \langle \text{本体} \rangle) \\
\langle \text{束縛リスト} \rangle &::= \ (\langle \text{識別子}_1 \rangle\ \langle \text{式}_1 \rangle) \ldots (\langle \text{識別子}_n \rangle\ \langle \text{式}_n \rangle)
\end{aligned}
$$

で定義される letseq 式は，変数を「順番に」宣言するものである．つまり，⟨識別子 $_i$⟩の有効範囲は，⟨式 $_{i+1}$⟩，...，⟨式 $_n$⟩ と ⟨式⟩ である．例えば以下のプログラム

```
(letseq ((x 4) (y (+ x 3)))
  (* x y))
```

の実行結果は 52 ではなく 28 である．letseq 式をインタプリタに実装・テストせよ．

3.6 mini Scheme^4 — 関数の導入

ここまでのところ，この言語には，いくつかのプリミティブ操作しか提供されておらず，mini Scheme プログラマが (プリミティブを組み合わせて) 新しい操作を定義することはできなかった．mini Scheme^4 では，関数を定義する機能と，定義された関数を適用する機能を提供する．

3.6.1 関数式と適用式

まずは，mini Scheme^4 の文法を示す．

⟨式⟩ ::= ...
| (lambda (⟨識別子 $_1$⟩ ... ⟨識別子 $_n$⟩) ⟨式⟩)
| (⟨式 $_0$⟩ ... ⟨式 $_n$⟩)

構文的な部分に関するインタプリタ・プログラムは，図 10 に示す．

lambda 式は，⟨識別子 $_1$⟩，...，⟨識別子 $_n$⟩ をパラメータとして，⟨式⟩ を評価する関数を表す．パラメータとして宣言される，⟨識別子 $_1$⟩，...，⟨識別子 $_n$⟩ の有効範囲は関数本体の⟨式⟩全体である．関数の適用はプリミティブ適用と似た構文を持ち，⟨式 $_1$⟩, ..., ⟨式 $_n$⟩の値を実引数として関数⟨式 $_0$⟩を呼び出す．

例えば，

```
(let ((f (lambda (x y) (* (+ x y) 3))))
  (f i x))
```

のようなプログラムが書ける．また，lambda 式は，式が必要な場所のどこにでも書くことができるので，上のプログラムを

```
((lambda (x y) (* (+ x y) 3)) i x)
```

などと書くこともできる．また，関数は他の関数への引数として渡すこともできる．

```
(let ((twice (lambda (f x) (f (f x))))
      (square (lambda (x) (* x x))))
  (twice square 5))
```

は，5 の自乗の自乗を計算する．

syntax.ml:

```
type exp =
   ...
  | Lambda of id list * exp
  | App of exp * exp list
```

parser.mly:

```
%token LAMBDA

Exp :
   ...
  | LPAREN LAMBDA LPAREN IDlist RPAREN Exp RPAREN { Lambda ($4, $6) }
  | LPAREN Exp Explist RPAREN { App ($2, $3) }

...
IDlist :
 /* empty */ { [] }
  | ID IDlist { $1 :: $2 }

Explist :
 /* empty */ { [] }
  | Exp Explist { $1 :: $2 }
```

lexer.mll:

```
let reservedWords = [
   ...
  ("lambda", Parser.LAMBDA);
]
```

図 10: 関数と適用 (1)

core.ml:

```
    let apply_prim p args =
      match p, args with
      | (Plus, [IntV i; IntV j]) -> IntV (i + j)
      | (Plus, [_; _]) ->
          failwith "One of the arguments is not an integer: +"
      | (Plus, _) -> failwith "Arity mismatch: +"
       ...

    let rec eval_exp env = function
       ...
      | If (e1, e2, e3) ->
          (match eval_exp env e1 with
            IntV test ->
              if test <> 0 then eval_exp env e2
              else eval_exp env e3
          | _ -> failwith "test expression is not an integer")
       ...
```

図 11: 関数と適用 (2)

3.6.2 値の定義の拡張

さて，上の例のように，mini Scheme[4] においては，関数は式を評価した結果となるだけでなく，変数の束縛対象にもなる**第一級の値**(*first-class value*) として扱う．そのため，expressed value, denoted value ともに関数値を加えて拡張する．

$$\begin{aligned} \text{Expressed Value} &= \text{整数}\,(\ldots,-2,-1,0,1,2,3,\ldots) + \text{関数値} \\ \text{Denoted Value} &= \text{整数} + \text{関数値} \end{aligned}$$

上の式で + は「または」を示している．関数値が具体的にどのように表現されるかは，ひとまず置いておいて，これを Objective Caml プログラムとして表現すると，以下のような定義になる．

```
type exval =
    IntV of int
  | ProcV of (* discussed later *)
and dnval = exval
```

値の種類が増えるため，インタプリタ内で値を直接扱う部分，つまりプリミティブの処理，if 式の処理では，プリミティブの引数や if 式のテスト部分を評価した値が整数であるかをチェックする必要がでてくる．これに関する主な変更点を図 11 に示す．

apply_prim では，各々のプリミティブについて，引数の数は正しいが，引数 (のいくつか) が整数で無い場合の処理を追加する．図 11 では Plus の場合のみを記述したが，その他のプリミティブも同様に書かれる．また，eval_exp では，コンストラクタ If の処理で，e1 の評価結果が整数であるかを明示的にチェックするコードになっている．

3.6.3 関数閉包と適用式の評価

さて，lambda 式の値をどのように表現するかを考える．lambda 式が適用された時には，パラメータを実引数 (の値) に束縛し，関数本体を評価するので，少なくともパラメータの名前と，本体の式が必要である．しかし，一般的には，以下の例のように，関数本体の式にパラメータ以外の変数 (つまり自由変数) が出現することも可能である．

```
(let ((x 2))
  (let ((addx (lambda (y) (+ x y))))
    (addx 4))
```

この addx の適用を行う際には，x の静的束縛を実現するために，本体中の x が 2 であることを記録しておかなければならない．つまり，一般的に関数が適用される時には，

1. パラメータ名
2. 関数本体の式，に加え
3. (少なくとも) 本体中のパラメータで束縛されていない変数 (自由変数) の束縛情報 (名前と値)

が必要になる．この 3 つを組にしたデータを**関数閉包・クロージャ**(*function closure*) と呼び，これを lambda 式の値として用いる．ここで作成するインタプリタでは，本体中の自由変数の束縛情報として，lambda 式が評価された時点での環境全体を使用する．これは，本体中に現れない変数に関する余計な束縛情報を含んでいるが，もっとも単純な関数閉包の実現方法である．

以上を元に残りの `core.ml` への変更点は図 12 のようになる．式の値には，環境を含むデータである関数閉包が含まれるため，`env` と `exval` は相互再帰的に定義されなければならない (`and` キーワード)．関数値は `ProcV` コンストラクタで表され，上で述べたように，パラメータ名のリスト，本体の式と環境の組を保持する．`eval_exp` で `Lambda` を処理する時には，その時点での環境，つまり `env` を使って関数閉包を作っている．適用式の処理は，適用される関数の評価，実引数の評価を行った後，本当に適用されている式が関数かどうかのチェック，実引数とパラメータの数が同じかどうかのチェックをして，本体の評価を行っている．本体の評価を行う際の環境 `newenv` は，関数閉包に格納されている環境を，パラメータ・実引数で拡張して得ている．

Exercise 3.10 [**必修課題**] mini Scheme[4] インタプリタを作成し，テストせよ．

Exercise 3.11 [$\star\star$] このインタプリタは，プリミティブと宣言された関数を区別しているので，

```
(let ((twice (lambda (f x) (f (f x)))))
  (twice add1 3))
```

core.ml:

```
    type exval =
        IntV of int
      | ProcV of id list * exp * env
    and dnval = ...
    and env = ...

    let rec eval_exp env = function
       ...
      | Lambda (ids, body) -> ProcV (ids, body, env)
          (* closed in the current env *)
      | App (rator, rands) ->
          let proc = eval_exp env rator in
          let args = eval_rands env rands in
          (match proc with
            (* check the operator is a procedure value *)
            ProcV (ids, body, env) ->
            if List.length ids = List.length args then
                (* extend the env in the closure *)
                let newenv = extend_env ids args env in
                eval_exp newenv body
              else failwith "# of parameters and arguments don't agree"
          | _ -> failwith "Applying a non-procedure value")
```

図 12: 関数と適用 (3)

のように，プリミティブを関数への引数として渡したりすることができない．(そもそもこれは構文的に mini Scheme[4] プログラムですらない．) プリミティブも第一級の値として扱えるようにインタプリタを改造・テストせよ．(ヒント: プリミティブを表現する expressed value を用意する．また，プリミティブの名前は構文解析時のキーワードとして扱わず，ただの識別子として導入し，大域環境で束縛する．)

Exercise 3.12 [⋆⋆] 以下は，加算を繰り返して 4 による掛け算を実現している mini Scheme[4] プログラムである．これを改造して，階乗を計算するプログラムを書け．

```
(let
  ((makemult
    (lambda (maker x)
      (if (= x 0) 0 (+ 4 (maker maker (sub1 x)))))))
  (let ((times4 (lambda (x) (makemult makemult x))))
    (times4 3)))
```

Exercise 3.13 [⋆⋆] mini Scheme[3] を導入した際に，静的束縛と動的束縛について述べた．動的束縛の下では，関数本体は，lambda 式を評価した時点ではなく，関数呼び出しがあった時点での環境をパラメータ・実引数で拡張した環境下で評価される．例えば，

```
(let ((a 3))
  (let ((p (lambda (x) (+ x a)))
        (a 5))
    (* a (p 2))))
```

というプログラムで，関数本体中の a は 3 ではなく 5 に束縛される．インタプリタを改造し，動的束縛 (dynamic binding) を実現せよ．

Exercise 3.14 [⋆] 動的束縛の下では，mini Scheme[5] で導入するような再帰のための特別な仕組みや，Exercise 3.12 のようなトリックを使うことなく，再帰関数を定義できる．以下のプログラムを静的束縛・動的束縛の下で実行し，結果について説明せよ．

```
(let ((fact (lambda (n) (add1 n))))
  (let ((fact (lambda (n)
          (if (= n 0) 1
              (* n (fact (sub1 n)))))))
    (fact 5)))
```

3.7 mini Scheme[5] — 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．mini Scheme[5] では，このよ

うな再帰的定義の機能を導入する．ただし，多くの言語と同様，単純化のため再帰的定義の対象を関数に限定[5]する．

まず，再帰的定義のための構文 letrec 式を，以下の文法で導入する．

$$
\begin{aligned}
\langle \text{式} \rangle \quad &::= \quad \ldots \\
&\quad | \quad \text{(letrec (}\langle \text{再帰関数定義リスト} \rangle\text{) } \langle \text{本体式} \rangle\text{)} \\
\langle \text{再帰関数定義リスト} \rangle \quad &::= \quad \text{(}\langle \text{識別子}_1 \rangle \text{ (lambda (}\langle \text{識別子}_{11} \rangle \ldots \langle \text{識別子}_{1n_1} \rangle\text{) } \langle \text{式}_1 \rangle\text{))} \\
&\qquad \ldots \text{ (}\langle \text{識別子}_m \rangle \text{ (lambda (}\langle \text{識別子}_{m1} \rangle \ldots \langle \text{識別子}_{mn_m} \rangle\text{) } \langle \text{式}_m \rangle\text{))}
\end{aligned}
$$

letrec 式は，$\langle$ 識別子$_1\rangle$，...，$\langle$ 識別子$_n\rangle$という名前の相互再帰的関数を示す変数を宣言し，その下で$\langle$ 本体式 $\rangle$ を評価する．let 式と違い，各変数が束縛される対象は lambda 式でなければならない．また，各変数の有効範囲は，$\langle$ 本体式 $\rangle$および，全ての lambda 式であるため，各 lambda 式の本体で，$\langle$ 識別子$_1\rangle$，...，$\langle$ 識別子$_n\rangle$を参照することが許される．

例えば，以下のプログラムは，階乗を計算する関数 fact を letrec を用いて宣言し，5! を計算するものである．

```
(letrec
  ((fact
     (lambda (n)
       (if (= n 0) 1
           (* n (fact (sub1 n)))))))
  (fact v))
```

この構文の基本的なセマンティクスは let 式と似ていて，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．各変数の有効範囲は，宣言される関数式を含むため，それらの関数の閉包を作るときの環境は，本体を評価するときの環境と同じもの，つまり，**これから得られるはずの拡張された環境**でなくてはならない．これを扱うために，ここでの実装は，Objective Caml の配列の更新機能を利用する．

図 13 が，`parser.mly`，`lexer.mll` を除く，プログラムの変更点である．`eval_exp` の `Letrec` を処理する部分は，`Let` の処理とほとんど同じである．環境に関する新しい操作 `extend_env_rec` が重要な部分である．`extend_env_rec` は，同時に宣言される変数名・関数 (パラメータリストと本体式の組)・環境を受け取る．最初に宣言される `vec` は，後で関数閉包が格納される配列で，初期値として，ダミーの整数値をいれている．`newenv` は，`vec` と宣言される変数名で拡張した新しい環境である．次は

```
(int -> 'a -> unit) -> 'a list -> unit
```

型を持つ補助関数 `iteri` の宣言で，

```
iteri f [a_0; a_1; ... a_n]
```

[5]Scheme や Objective Caml などでは，(構文的には) 任意の式を再帰的定義の対象にできる．

syntax.ml:

```
    type exp =
      ...
      | Letrec of (id * (id list * exp)) list * exp
```

core.ml:

```
    let extend_env_rec syms procs env =
      (* assumes List.length syms = List.length procs *)
      let vec = Array.make (List.length syms) (IntV 0) in
      let newenv = ExtendEnv (syms, vec, env) in
      let rec iter f i = function
          [] -> ()
        | x :: ls -> f i x; iter f (i + 1) ls in
      let iteri f = iter f 0 in
      iteri
        (fun i (ids, body) -> vec.(i) <- ProcV (ids, body, newenv))
        procs;
      newenv

    let rec eval_exp env = function
      ...
      | Letrec (procdefs, body) ->
          let (ids, procs) = List.split procdefs in
          let newenv = extend_env_rec ids procs env in
          eval_exp newenv body
```

図 13: 再帰的関数定義

は

`f 0` a_0`; f 1` a_1`; ...; f n` a_n

を実行する．これを使って，newenv を，関数閉包を束縛するように更新している．関数閉包は，拡張された環境 newenv を使って作られていることに注意．最終的には，更新された環境を返している．

Exercise 3.15 [**必修課題**] 図に示した syntax.ml にしたがって，parser.mly と lexer.mll を完成させ，mini Scheme[5] インタプリタを作成し，テストせよ．ただの再帰的関数だけでなく，相互再帰的な関数もテストすることが望ましい．

Exercise 3.16 [⋆⋆⋆] letrec を使って定義されるものは，lambda 式に限られていたが，この制限をどの程度緩められるか議論し，インタプリタを改造せよ．

3.8 リスト

Exercise 3.17 [⋆⋆] リスト値が扱えるように mini Scheme[5] インタプリタを変更せよ．Scheme のリストは空リストを ()，リストへの要素の追加を (2 引数) プリミティブ cons，要素を列挙しリストを構成するための (可変個引数) プリミティブ list で構成する．また，リストが空かどうかを判定する null，先頭要素を返す car，後続リストを返す cdr というプリミティブを使ってリスト操作を行なう．

リストの出力結果は要素を空白で区切って並べ () で囲んだ形になる．

```
⇒ (list 1 2 3 4)
(1 2 3 4)
⇒ (cons 0 (list 1 2 3 4))
(0 1 2 3 4)
⇒ (cons (list 1 2) (list 3 4))
((1 2) 3 4)
```

最後の例からわかるように，Scheme では要素の型が揃っている必要はないことに注目．(第一要素がリスト (1 2) であり，残りの要素は整数である．) これで，リスト操作をする様々な関数をテストせよ．

```
⇒ (letrec ((append (lambda (l1 l2)
                     (if (null l1) l2
                         (cons (car l1) (append (cdr l1) l2))))))
    (append (list 1 2) (list 3 4)))
(1 2 3 4)
```

Exercise 3.18 [⋆⋆] Scheme では，リスト要素の型が揃う必要がないばかりか，cons の第二引数がリストである必要すらない．実は，Scheme のリストは (cons で作られる) ペア構造の特別な場合として扱われる．つまり，リストは空リストもしくは，リストを第二要素とする

ペアとして定義される．このような cons を実装せよ．(この場合正確には car, cdr はそれぞれ，ペアの第一要素・第二要素を返すプリミティブである．)

以下のように，第二要素がペアでないペアは . を使って表示する．

```
⇒ (cons 1 2)
(1 . 2)
⇒ (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
⇒ (cons (cons 1 2) 3)
((1 . 2) . 3)
⇒ (cdr (cons 1 2))
2
```

Exercise 3.19 [⋆⋆⋆] ML に見られるようなパターンマッチ構文を設計し実装せよ．

Exercise 3.20 [⋆⋆] Scheme では，**引用**(*quotation*) といって，(リストなどの) データ値の外部表現 (インタプリタの表示結果) をプログラム中に直接記述して，cons や list などのプリミティブを使うことなく構成することができる．具体的には外部表現の前に引用符 ' をつけることで，外部表現から値を構成することができる．

```
⇒ '1
1
⇒ '(1 2)
(1 2)
⇒ (cons 3 '(1 2))
(3 1 2)
⇒ (cons '(4 3) '((5 6) 7))
((4 3) (5 6) 7)
⇒ (cdr '(1 2 3))
(2 3)
```

このような引用機構を実装せよ．ここでは，引用されるものは，整数，リスト，(前の問題をやっている場合) ペアとする．

3.9 mini Scheme[6] — 変数への代入の導入

次に，変数への代入のための構文 set 式を，以下の文法で導入する．

⟨式⟩ ::= ...
| (set ⟨識別子⟩ ⟨式⟩)

set 式は ⟨識別子⟩ の変数に ⟨式⟩ の値を「代入」する．set 式自体の値は 0 と定義する．

さて,「代入」とはなんだろうか？代入について考える前に，束縛の意味をもう一度吟味してみよう．プログラミング言語 (や論理) における束縛とは，何か (ここでは denoted value) に名前をつけるための仕組みであり，一旦束縛関係が発生したら，通常その関係は束縛の有

効範囲内で不変である．set のような代入を導入した言語においても，その原則は保たれるとしたほうが，様々な言語を統一的に捉えることができる．束縛関係が不変だとした場合，set は，以下のようなものとして考える．まず，変数は式の値そのものにではなく,「値を格納している場所 (メモリアドレス)」に束縛されると考える．この変数と「場所」の束縛関係はこの変数の有効範囲内で不変である．そして，set 式は変数が束縛されたアドレスの示す「内容」を⟨式⟩の値に変更するものである，と考える．このように考えることによって束縛の概念に手をつけずに代入を扱うことができる．以上の議論を expressed value, denoted value の定義に反映させると，

$$\begin{aligned} \text{Expressed Value} &= \text{整数}\,(\ldots,-2,-1,0,1,2,3,\ldots) + \text{関数値} \\ \text{Denoted Value} &= (\text{整数} + \text{関数値})\ \text{への参照}\ (\text{アドレス}) \end{aligned}$$

のようになる．

このようにした言語では，プログラム中の変数参照は二種類の意味を持つことになる．例えば

```
(set x (add1 x))
```

という式において，最初の x の出現の示すものは，変数の denoted value，すなわち束縛されているアドレスであり，ふたつめの出現の示すものは，束縛されているアドレスの「内容」(expressed value) である．このように，代入のある言語では，変数に「二種類の値」が関連づけられていると考え，アドレスとしての値を**左辺値**(*L-value*)，その内容としての値を**右辺値**(*R-value*) と呼ぶことがある．これらの名前は，C 言語の `x=x+1;` といった代入文で，左辺の x はアドレスであり，右辺の x はその内容を示すことに由来する．これに対し，Objective Caml の参照型は，これらの概念をきっちりと分けており，x が `int ref` 型とすると，x と書いた場合は常にその「左辺値」を示し，x の「右辺値」を参照する際には常に `!x` と明示的に書かなければならない．

代入は，プログラムの別の個所でなにかを共有するために用いることができる．以下のプログラムは，inc，get という関数で，counter を共有している．

```
(let ((counter 0))
  (let ((get (lambda () counter))
        (inc (lambda ()
               (let ((d (set counter (add1 counter)))) 0))))
    (let ((d (inc)))
      (let ((d (inc)))
        (get)))))
```

d はダミーの変数で，let とともに用いて式の実行の順序付けをしている．counter の内容を 1 増やす関数 inc を 2 回呼んで，counter の内容を get で得ており，全体の評価結果は 2 になる．このふたつの関数は，値をパラメータとしてやり取りするのではなく，変数の状態を変更することで行っている．また，counter を関数にプライベートなものとして宣言することで，隠れた状態を実現できる．

```
(let ((inc_and_get
       (let ((counter 0))
         (lambda ()
           (let ((d (set counter (add1 counter))))
             counter)))))
  (+ (inc_and_get) (inc_and_get)))
```

2 回の inc_and_get の呼出しは，呼出し時点の内部状態 (counter の値によって) それぞれ違う値を返している．

mini Scheme[6] では，関数呼び出し毎に，各パラメータのために，実引数を格納するためのアドレスを新たに用意する．このような仕組みを，**値呼び出し**(*call-by-value*) と呼ぶ[6]．値呼び出しの下では，関数のパラメータに対する代入は関数の外側では観察できない．例えば，

```
(let ((x 100))
  (let ((p (lambda (x)
             (let ((d (set x (add1 x)))) x))))
    (+ (p x) (p x))))
```

の値は 202 である．つまり，p の呼び出し毎に，新しいアドレスが実引数のために割り当てられ，そこに set で代入されても，p の外側の x には影響をおよぼさないのである．

さて，プログラムの主な変更点を図 14 に示す．まず，`dnval` 型の定義を見るとわかるように．参照を表現するものとして `ref` 型を用いる．変更後の `apply_env` は `dnval` を返すので，変数参照の `Var` を処理するときには，`!` オペレータを使って，中身の `exval` を取り出す必要がある．代入文 `Assign` の処理は，代入される式の値と，変数の denoted value を計算し，Objective Caml の代入 `:=` を使って更新している．`eval_rands` 関数は，引数を評価した後，それを新しく割り当てた参照値に格納している．プリミティブの引数では，この操作は必要なく，引数式の expressed value をそのまま渡せばよいので，以前の `eval_rands` の定義を `eval_prim_rands` として使っている．

Exercise 3.21 [⋆] 図に示した `syntax.ml` にしたがって，`parser.mly` と `lexer.mll` を完成させ，mini Scheme[6] インタプリタを作成し，テストせよ．

Exercise 3.22 [⋆] 式を順番に評価するための begin 式を導入したインタプリタを作成せよ．

$$
\begin{aligned}
\langle \text{プログラム} \rangle &::= \langle \text{式} \rangle \\
\langle \text{式} \rangle &::= \ldots \\
&\quad | \quad (\text{begin } \langle \text{式}_1 \rangle \ldots \langle \text{式}_n \rangle)
\end{aligned}
$$

begin 式は，$\langle$ 式 $_1\rangle$から順に評価し，$\langle$ 式 $_n\rangle$ の値を全体の式の値とする．これを用いて，(let を使って評価順序をつけているような) テキストの例を書き直し，インタプリタでテストせよ．

[6]値呼び出しには，もうひとつ，関数呼び出しの実引数式は関数が呼び出される前に値に評価する，という意味もある．

syntax.ml:

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

core.ml:

```
type exval = ...
and dnval = exval ref
...

let extend_env_rec syms procs env =
  let vec = Array.make (List.length syms) (ref (IntV 0)) in
  ...
  iteri
    (fun i (ids, body) -> vec.(i) <- ref (ProcV (ids, body, newenv)))
    procs;

let rec eval_exp env = function
   ...
  | Var sym -> !(apply_env sym env)
  | Prim (p, es) ->
      let args = eval_prim_rands env es in
      ...
   ...
  | Assign (id, exp) ->
      let arg = eval_exp env exp in
      let idref = apply_env id env in
      begin idref := arg; IntV 0 end

and eval_rands env = function
    [] -> []
  | e :: rest -> ref (eval_exp env e) :: eval_rands env rest

and eval_prim_rands env = function
    [] -> []
  | e :: rest -> eval_exp env e :: eval_prim_rands env rest
```

図 14: 変数への代入

Exercise 3.23 [⋆⋆] 言語に代入を導入するもうひとつの方法は，Objective Caml に見られるように,「値の格納場所」を expressed value として導入し，代入だけでなく，格納場所の作成・値の取りだしも明示的な操作として導入する方法である．(引数を初期値とした) 値の格納場所を作成する ref，引数として与えられた場所の内容を取り出す deref，第 1 引数として与えられた場所に，第 2 引数の値を代入する setref をプリミティブとして導入せよ．値の定義は以下に与える．

$$\begin{aligned} \text{Expressed Value} &= \text{整数}\,(\ldots,-2,-1,0,1,2,3,\ldots) + \text{関数値} \\ &\quad + \text{Expressed Value への参照} \\ \text{Denoted Value} &= \text{Expressed Value} \end{aligned}$$

そして，以下のプログラムをテストせよ．

```
(let
  ((g (let ((count (ref 0)))
        (lambda ()
          (let ((d (setref count (add1 (deref count)))))
            (deref count))))))
  (+ (g) (g)))
```

3.10 リストの更新

Exercise 3.24 [⋆⋆] setcar, setcdr というキーワードを追加し, Exercise 3.17 で導入したリスト構造の先頭要素 (car) と後続部分 (cdr) の値を更新できるようにインタプリタを変更せよ. (注意: この問題は, Exercise 3.18 のようにリストをペア構造として扱う方が遥かに見通しがよいので, Exercise 3.18 を行なってから解答した方がよい.)

setcar は引数として二つの式をうけとり, 代入のように, 副作用として第一式が示すリストの先頭要素を第二式が示す値で置き換える．同様に, setcdr は引数として二つの式をうけとり, 第一式が示すリストの後続要素を第二式が示すリストで置き換える．返り値については特に限定しない (代入のときを参考にせよ).

```
⇒ (let ((xs (list 0 2 3 4)))
      (let ((d (setcar xs 1)))
        xs))
(1 2 3 4)
⇒ (let ((xs (list 0 1 3 4)))
      (let ((d (setcdr (cdr xs) (list 2 3 4))))
        xs))
(0 1 2 3 4)
```

setcar, setcdr は, Exercise 3.18 のようにリストをペア構造としてみれば,

```
⇒ (let ((p (cons 0 2)))
      (let ((d (setcar p 1)))
```

```
          p))
(1 . 2)
⇒ (let ((p (list 0 1 2)))
      (let ((d (setcdr p 4)))
          p))
(0 . 4)
```

のように, それぞれペアの第一要素, 第二要素を置き換えるものとして実現できる．setcar, setcdr を用いたプログラムを書いてテストせよ.

```
⇒ (let ((xs (list 1 3 4 5)))
      (let ((d (setcdr xs (cons 2 (cdr xs)))))
          xs))
(1 2 3 4 5)
⇒(letrec ((nconc (lambda (xs ys)
                  (if (null xs) ys
                      (if (null (cdr xs)) (setcdr xs ys)
                          (nconc (cdr xs) ys))))))
  (let ((a (list 0 1 2))
        (b (list 3 4 5)))
    (let ((d (nconc a b)))
      a)))
(0 1 2 3 4 5)
```

Exercise 3.25 [⋆⋆] 前問で導入した setcdr を用いて,

```
⇒ (letseq ((xs (cons 1 ()))
           (d (setcdr xs xs)))
      xs)
```

というプログラムを実行すると, 無限に要素 1 が続く無限リストが返るはずである．これは, リスト l の cdr 部が l 自身を参照するという, 循環参照が起こっているからである．このような循環参照による無限リストを「何らかの (無限ループしない) 手段で」表示できるように `Core.pp` の定義を工夫せよ．(どのくらい凝ったことをするかで難易度は大きく変わる．)

3.11 いろいろな引数渡しの方式 — 値呼び・参照呼び・名前呼び・必要呼び

前節までの言語では，値呼び出しに基づいて関数呼び出しの際の引数渡しを実現してきた．本節では，FORTRAN, Pascal などに見られる **参照呼出し**(*call-by-reference*) と，**遅延評価**(*lazy evaluation*) を使った，Algol60 に見られる **名前呼出し**(*call-by-name*)，Haskell などに見られる，**必要呼出し**(*call-by-need*) といった引数渡しの方法をみていく.

3.11.1 参照呼出し

前節で説明したように，値呼出しの下では，関数のパラメータに対する代入は関数呼出し側に影響をおよぼすことはない．つまり，関数呼出しの前後で，ある変数の内容が変わるこ

とがなく，呼出し側・関数側の干渉がない[7]ことが保証されるため，プログラムの動作を考えるに当たって，非常に助けになる．

しかし，一方で，関数に変数を渡し，内部で代入をしてもらい，その「代入された結果」を呼出し側で利用したい場合もある．これは，関数呼出し時に，実引数の格納場所をそのまま関数に渡すことで実現できる．このような引数渡しの方法を参照呼出しという．関数への引数が，変数参照である場合，その格納場所 (denoted value) を渡し，他の種類の式である場合は，値呼出しと同様に，新たに格納場所を確保し，その引数の値 (expressed value) を割り付けて渡す．

参照呼出しは，しばしば，複数の値を返す関数を模倣するために用いられる．ひとつの値を関数の通常の返り値とし，残りを代入を使って渡すのである．例えば，Exercise 3.23 のプログラムは，参照呼出しの下で

```
(let
  ((a 3)
   (b 4)
   (swap (lambda (x y)
           (let ((temp x))
             (let ((d (set x y)))
               (set y temp))))))
  (let ((d (swap a b)))
    (- a b)))
```

と書くことができる．これは値呼出しの下では -1 を，参照呼出しの下では swap 内の代入が呼出し側に及ぶため，a と b の内容が入れ替わって，1 となる．

参照呼出しの引数は，同じ場所を指す可能性がある．次のプログラムで，引数 x，y は同じ場所を指すので，実行結果は 4 になる．

```
(let
  ((b 3)
   (p (lambda (x y)
        (let ((d (set x 4))) y))))
  (p b b))
```

このように，違う名前が同じものを指す現象を**エイリアシング**(*aliasing*) と呼ぶ．通常，ある変数の内容を更新して別の名前の変数の内容に影響がでることは期待しないため，エイリアシングはプログラムの挙動を非常に分かりにくくする．(もちろん，値呼出しであっても，参照が expressed value であるような言語ではエイリアシングが発生する．)

参照呼出しの実現は図 15 にあるように，実は簡単である．まず，値の定義は前と変わらず，denoted value は expressed value への参照として定義される．インタプリタの変更点は，関数呼出しの引数を評価する時に `ref` を作る部分のみである．`eval_rands` には，引数が変数であった場合に，`apply_env` で得られた denoted value をそのまま使うようにしている．また，let 式に関しては値呼出しを使うので，let 用の `eval_let_rands` を定義している．

[7] Exercise 3.23 のような機能がある場合，もちろんこの限りではない

core.ml:

```
    let rec eval_exp env = function
       ...
      | Let (bs, e) ->
          ...
          let arg_vals = eval_let_rands env args in
          ...

    and eval_rands env = function
       ...
      | Var id :: rest -> (apply_env id env) :: eval_rands env rest
       ...

    and eval_let_rands env = function
        [] -> []
      | e :: rest -> ref (eval_exp env e) :: eval_let_rands env rest
```

図 15: 参照呼出し

Exercise 3.26 [⋆] 参照呼出しをインタプリタに実装し，上の swap の例をテストせよ．また，`eval_let_rands` を `eval_rands` で代用すると，どうなるだろうか．

Exercise 3.27 [⋆⋆] Pascal などでは，関数を定義する際に，パラメータ毎に値呼出しを使うか参照呼出しを使うかを指定することができる．この仕組みを，文法からデザインし実装・テストせよ．

3.11.2 遅延評価と call-by-need, call-by-name

次に，参照呼出しとはかなり違う形態の引数渡しの方法についてみる．関数定義において，本体中でパラメータのいくつかを使用しないことがありえる．このとき，呼出し側で対応する実引数を評価するのは無駄である．実引数の評価が止まらない場合には，問題ですらある．

mini Scheme のように，関数が第一級の値である場合には，無駄な引数を評価しないために，パラメータ無しの関数として渡し，関数内部で使うときに初めて関数適用を行って引数の評価を行うというプログラミングで，この問題を避けることができる．このように，評価を遅らせるために用いられるパラメータ無しの関数を *thunk* と呼び，thunk を構成・評価することを，それぞれ *freezing*, *thawing* と呼ぶ．例えば，

```
(let ((p (lambda (x y) (* 2 x))))
  (p (+ 2 4) (fact 150)))
```

のようなプログラムを，機械的に実引数を freeze し，パラメータが使われているところで thaw するようにして，以下のように

```
(let ((p (lambda (x y) (* 2 (x))))
      (p (lambda () (+ 2 4)) (lambda () (fact 150))))
```

のように書き換えることができる．(ここでは fact は階乗を計算するプリミティブと考えよ．) これにより，結果が使われないが時間のかかる階乗の計算をせずに，プログラムが実行される．

いくつかの言語では，引数の計算を，その計算結果が使われるまで (自動的に) 遅らせる**遅延評価**(*lazy evaluation*) の仕組みが備わっている．遅延評価の仕組みは，パラメータが複数回現れたときの処理の違いによって 2 種類に分けられる．もっとも単純なやり方は，引数が使われる度に，thunk を thaw してして値を求めるもので，**名前呼出し**(*call-by-name*) と呼ばれ，Algol60 などで使われた．しかし，代入などの副作用が無い場合，くりかえし計算するのは同じ値が得られるだけで無駄である．これを改良したのが，Haskell などで採用された **必要呼出し**(*call-by-need*) という仕組みで，一度 thaw した thunk はその値を覚えておいて，二度目以降に使われるときには，覚えておいた値を使うものである．このふたつの機構は副作用がない限り，同じ実行結果になる．しかし，代入などを使うと，このふたつの違いを見ることができる．例えば，以下のプログラム

```
(let
  ((g (let ((count 0))
        (lambda ()
          (let ((d (set count (add1 count))))
            count))))
   (double (lambda (x) (+ x x))))
  (double (g)))
```

で，g は呼び出される度に，前回より 1 大きい値を返す．名前呼出しの下では，g は，x の出現回数である 2 回呼び出されて，3 を返す．一方，必要呼出しの下では，最初の x について g が呼び出され 1 が値となる．しかし，次の x の出現では，呼出しが起らず，前回と同じ 1 に評価され，全体の値は 2 になる．

遅延評価言語は，副作用の無い場合，プログラムの挙動の推論を非常に単純なやり方でできるという利点がある．つまり，関数適用は，本体内のパラメータを引数で「そのまま」置き換えることでモデル化できる．(値呼出しの場合，引数の評価が止まらない場合もあるので，厳密にはこのような推論はできない．) また，データ構造と遅延評価を組み合わせると，無限の構造を持つデータなども簡潔に表現することができる．一方，遅延評価はプログラムの制御の流れ・実行順序といったものをわかりにくくするため,「いつ」おこるかが重要である，副作用と組み合わせると，プログラムの挙動が著しく分かりにくくなる．実際，遅延評価が採用されている多くの言語は，副作用を起こす機能がないものが多い．

さて，ここでは名前呼出しを練習問題として，必要呼出しを実装していく．変数には，thunk が束縛される場合もあるため，

$$\text{Expressed Value} \quad = \quad \text{整数}\,(\ldots,-2,-1,0,1,2,3,\ldots) + \text{関数値}$$

Denoted Value = (Expressed Value + *thunk*) への参照

とする．`core.ml` の主要な変更点を図 16 に示す．まず，`dnval` は，`pre_dnval` への参照として定義されている．`pre_dnval` は，thunk もしくは，すでに thaw された expressed value である．thunk は本質的にはパラメータのない関数閉包であり，本体と環境を保持している．

`eval_exp` は変数参照の部分が，大きな変更点である．まず，環境から変数の束縛された denoted value を取り出す．もしも，その内容が thunk であった場合は，thunk の thawing を行う．しかも，必要呼出しなので，一度評価した thunk の値で変数の内容を更新する．

`eval_rands` は，関数呼出しの引数を処理する部分である．これもこれ以上評価が進まない，整数リテラル・lambda 式の場合を除いて，thunk を生成して，引数の評価を遅らせている．整数リテラル・lambda 式はそのまま値になるので，thunk を作らず最初から thaw されたものになる．

Exercise 3.28 [⋆] 他に必要な `core.ml` の変更を施して，必要呼出しインタプリタを実装・テストせよ．

Exercise 3.29 [⋆] 必要呼出しインタプリタを改造し，名前呼出しの仕組みを実装・テストせよ．

Exercise 3.30 [⋆⋆] 上で実装されたインタプリタは let に関しては，定義される値の遅延評価が行われない．let を遅延評価が行われるように修正せよ．また，遅延評価を行う let と，遅延評価を行わない strictlet を導入し，プログラム上で使い分けられるよう改造せよ．

参考文献

[1] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1997. 現在，関数型プログラミングの教科書の中で Caml を直接対象にした (英語では) 唯一のもの．

[2] R. Kent Dybvig. **プログラミング言語** *SCHEME*. ピアソン・エデュケーション, 2000.

[3] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, second edition, 2001.

[4] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.08: Documentation and user's manual*, 2004. `http://caml.inria.fr/ocaml/htmlman/index.html`.

[5] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997. Standard ML の形式的定義．数学的な定義が並んでいるもので解説はないので読むのは困難．コンパイラ実装者など言語仕様を正確に知りたい人向け．

core.ml:

```
...
and pre_dnval =
    Thunk of exp * env
  | Thawed of exval
and dnval = pre_dnval ref
...

let rec eval_exp env = function
   ...
  | Var sym ->
      let varref = apply_env sym env in
      (match !varref with
        Thunk (e, env') ->
          let v = eval_exp env' e in
          varref := Thawed v;
          v
      | Thawed v -> v)
   ...

and eval_rands env = function
   ...
  | ILit i :: rest -> ref (Thawed (IntV i)) :: eval_rands env rest
  | Lambda (ids, body) :: rest ->
      ref (Thawed (ProcV (ids, body, env))) :: eval_rands env rest
  | e :: rest -> ref (Thunk (e, env)) :: eval_rands env rest

   ...
and eval_let_rands env = function
    [] -> []
  | e :: rest ->
     ref (Thawed (eval_exp env e)) :: eval_let_rands env rest
```

図 16: 必要呼出し

[6] 湯淺 太一. *Scheme* **入門**. 岩波コンピュータサイエンス. 岩波書店, 1991. 絶版?