2006年度
「計算機科学実験及演習4(プログラム検証)」
実験資料
型推論によるプログラム解析

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，プログラミング言語 ML のサブセットの処理系 (インタプリタおよび型推論機構) の実装を通じて，**プログラムの実行前にエラーを検出する技術**の基本的な仕組みを理解することにある．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 (3 週程度) で，処理系を実装するための言語として，ML の方言である Objective Caml の演習を，実験時に配布するテキスト「Objective Caml 入門」を使って行う．ここでの Objective Caml の習得を通じ，ML プログラムの挙動，型推論に関する直感的理解を得る．そののち，インタプリタ・型推論器の作成を行う．ML は後に述べるように静的に強く型付けされた言語だが，まずは，(Scheme のように) 静的な型のない言語としてインタプリタを実装し，それに型推論機構を加えていく．

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〜第 3 週	「Objective Caml 入門」第 2〜7 章の演習
第 4 週〜第 5 週	インタプリタの実装
第 5 週〜第 6 週	型推論機構の実装

1.2 成績評価

前半の Objective Caml 演習 (40 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・型推論機構実装実験 (各 30 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 [3] (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，[2, 第 3 章] を参考にしている．(この本では Scheme 自身で Scheme インタプリタを作成している．)

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 [4] 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．

[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．

2.2 Objective Caml 言語の雰囲気

Objective Caml 言語の概要をプログラム例を混じえて紹介する．これは Objective Caml という言語自体の説明だけでなく，この実験で作成する処理系の動作例にもなっている．

Objective Caml は，いわゆる関数型言語である 計算の仕組みは Scheme に似ていて，式を値に評価することでプログラム実行が進む．例えば，Scheme では，

```
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))
(fact 5)
```

と入力すると，階乗関数 `fact` が定義されて，次の式の入力により 120 が返ってくるが，Objective Caml では以下のように書く．

```
let rec fact n =
  if n = 0 then 1 else n * fact (n-1);;
fact 5;;
```

何となく対応関係がわかるだろうか．

関数も値である もちろん，Scheme のように，関数を返す関数や，関数を引数に取る関数なども自由に使える．例えば，以下は与えられた関数 f と整数 n から $\sum_{i=0}^{n} f(i)$ を計算する関数である．

```
let rec sum (f, n) =
  if n = 0 then f(n) else f(n) + sum (f, n-1);;
sum (fact, 5);;
```

とすると，$0! + 1! + 2! + 3! + 4! + 5!$ が計算されて，154 が返ってくる．

Objective Caml には静的型があり，強く型付けされる Objective Caml プログラムに対しては実行前に型検査が行われ，演算子などの誤った使用がないかどうかが検査される．型の合わない計算をさせようとすると，**実行をする前に**(静的に) エラーが発生して誤りを指摘される．

```
let hoge x = x + 1 + "hoge";;
```

などと入力すると，その時点で，`hoge` を呼び出してもいないのに，

```
This expression has type string but is here used with type int
```

つまり，"hoge" という文字列が，+ という整数を期待している演算子の引数として使われていますよ，という意味のメッセージ (ここではシステムからの返答を表すために斜体を使っている) とともにコンパイルエラーが発生する．(実際は "hoge" の下に下線がひかれて，This expression がどれなのかも教えてくれる．) このエラーは，hoge を呼び出して実際に足し算が行われようとする前に発生するのがミソである．また，Objective Caml では，型検査を通過した場合には，文字列と整数の足し算のようなエラーが実行時に発生しないことが，(数学的に) 保証されている．このことを Objective Caml は**強く型付けされた**(*strongly typed*) 言語である，という．

Objective Caml は型推論をしてくれる C 言語にも静的型があるが，関数宣言の際に，仮引数の型を宣言する必要がある．これに対して Objective Caml では上の例からも見てとれるように，引数の型や返り値の型をいちいち書かなくても，システムが推論をしてくれる．例えば，

```
let rec fact n =
  if n = 0 then 1 else n * fact (n-1);;
```

とすると，システムからは，

```
val fact : int -> int = <fun>
```

という反応が返ってくる．これは fact が整数 (int) を受け取って，整数を返す関数 (->) だということを示している．

Objective Caml にはパターンマッチ機能がある Objective Caml には，パターンマッチという機能があって，リストなど構造のある値に対して，パターンを当てはめて，値の一部を取りだすことができる．Objective Caml では，リストを [] (空リスト)，:: (cons) を使って，1 :: 2 :: 3 :: [] のように書くのだが[2]，整数リストの先頭 2 要素までの和を計算する関数 sum_of_first_two は，パターンマッチを使うと

```
let sum_of_first_two l =
  match l with
     [] -> 0
   | x :: [] -> x
   | x :: y :: rest -> x + y;;
val sum_of_first_two : int list -> int = <fun>
```

と書ける． match は関数の引数 l に対して，[] (空リストパターン) や x::[] (1 要素パターン) や x::y::rest (2 要素以上パターン) を当てはめて， 当てはまったら，変数 x (や y) をリストの要素として，計算を行う．パターンマッチを使うと，条件分岐を複雑に入れ子にすることなく，全ての場合が一度に書き下せるのが魅力である．ちなみに，型に現れる -> の左側の int list は整数が並んだリスト が引数としてとれることを示している．

[2] [1; 2; 3] という記法もある

Objective Caml の型は多相的 多相的，というのは，ひとつのプログラムの型がいろいろ変化することを示している．例えば，以下は，リストの長さを計算する関数であるが，

```
let rec length list =
  match list with
     [] -> 0
   | x::rest -> 1 + length rest;;
```

これは，整数のリストに使うこともできるし，文字列のリストに使うこともできる．

```
length (2::3::4::[]);;
- : int = 3
length ("hoge"::"foo"::"bar"::"baz"::[]);;
- : int = 4
```

つまり，ひとつの関数 `length` を,「整数リストを受けとって整数を返す」関数や「文字列リストを受けとって整数を返す」関数という別の型のものとして使える．ソーティング関数なども，整数のソーティング，文字列のソーティングなど異なる種類のデータに対するソーティングを，ひとつの定義で記述することができる．このようなプログラミングを，型が多相的でない C 言語などで (安全性を損なわないで) 行うのは難しい．

その他の特徴 Scheme と同様，**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．また，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．

3 ML 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．本実験では,

$$\begin{aligned} \text{定義する言語} &= \text{Objective Caml} \\ \text{定義される言語} &= \mathsf{ML}(\text{Objective Caml のサブセット}) \end{aligned}$$

である．一般には，定義する言語と定義される言語は異なるが，今回のように両者が一致する場合，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．以下では，定義する言語でのプログラムの記述にはタイプライタ体 (abcde) を，定義される言語のプログラムにはサン・セリフ体 (abcde) を用いて，両者を区別する．

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 7 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．

`environment.mli`, `environment.ml` インタプリタ・型推論で用いる，環境と呼ばれるデータ構造を定義する．

`eval.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 ML[1] インタプリタのための7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 7 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 miniml 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) miniml を起動すると # というプロンプトが現れるので， ML プログラムを打つと結果が表示される．

```
> miniml
# x;;
val - = 10
# x + 3;;
val - = 13
```

(起動時に大域変数 x の値は 10 になっている．) 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 ML[1] インタプリタ — プリミティブ演算，条件分岐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，真偽値，条件分岐，加算乗算と変数の参照のみ (新しい変数の宣言すらできない!) を持つ言語 ML[1] から始める．

最初に，ML[1] の文法を以下のように与える．

$$
\begin{aligned}
\langle \text{プログラム} \rangle &::= \langle \text{式} \rangle ;; \\
\langle \text{式} \rangle &::= \langle \text{識別子} \rangle \\
&\quad | \quad \langle \text{整数リテラル} \rangle \\
&\quad | \quad \langle \text{真偽値リテラル} \rangle \\
&\quad | \quad \langle \text{式}_1 \rangle \langle \text{二項演算子} \rangle \langle \text{式}_n \rangle \\
&\quad | \quad \text{if} \langle \text{式}_1 \rangle \text{ then} \langle \text{式}_2 \rangle \text{ else} \langle \text{式}_3 \rangle \\
&\quad | \quad (\langle \text{式} \rangle) \\
\langle \text{二項演算子} \rangle &::= \quad + \mid * \mid <
\end{aligned}
$$

プログラムは，;; で終わるひとつの式である．式は，識別子による変数参照，整数リテラル，真偽値リテラル (true と false)，条件分岐のための if 式，または二項演算子式，括弧で囲まれた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，予約語ではない文字列である．この段階では予約語は if, then, else, true, false の 5 つである．例えば，以下の文字列はいずれも ML[1] プログラムである．

```
3;;
true;;
x;;
3 + x';;
(3 + x1) * false;;
```

また，+, * は左結合，結合の強さは，強い方から，*, +, <, if 式 とする．

それでは，構文に関する部分から順に，6 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．) `binOp`，`exp`，`program` 型に関しては上の文法構造を (括弧式を除いて) 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．

```
(* ML interpreter / type reconstruction *)
type id = string

type binOp = Plus | Mult | Lt

type exp =
    Var of id
  | ILit of int
  | BLit of bool
  | BinOp of binOp * exp * exp
  | IfExp of exp * exp * exp

type program =
    Exp of exp
```

図 1: ML[1] インタプリタ: syntax.ml

```
%{
  〈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，いちいち 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，入力の終了を示す “;;” に対応するトークン LPAREN, RPAREN, SEMISEMI と，プリミティブ (+, ∗, <) に対応するトークン PLUS, MULT, LT, 予約語 if, then, else, true, false に対応するトークンが宣言されている．(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．トークン名は全て英大文字としている．) こ

```
%{
open Syntax
%}
%token LPAREN RPAREN SEMISEMI
%token PLUS MULT LT EQ COLONCOLON
%token IF THEN ELSE TRUE FALSE

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

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

toplevel :
    Expr SEMISEMI { Exp $1 }

Expr :
    IfExpr { $1 }
  | LTExpr { $1 }

LTExpr :
    PExpr LT PExpr { BinOp (Lt, $1, $3) }
  | PExpr { $1 }

PExpr :
    PExpr PLUS MExpr { BinOp (Plus, $1, $3) }
  | MExpr { $1 }

MExpr :
    MExpr MULT AExpr { BinOp (Mult, $1, $3) }
  | AExpr { $1 }

AExpr :
    INTV { ILit $1 }
  | TRUE { BLit true }
  | FALSE { BLit false }
  | ID { Var $1 }
  | LPAREN Expr RPAREN { $2 }

IfExpr :
    IF Expr THEN Expr ELSE Expr { IfExp ($2, $4, $6) }
```

図 2: ML[1] インタプリタ: parser.mly

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．字句解析プログラムは文字列を読み込んで，この型の値(の列)を出力することになる．

- %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．

- %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ML^{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．

図2の文法規則が，上で述べた結合の強さ，左結合などを実現していることを確かめてもらいたい．

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⟩
...

[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．

```
rule 〈エントリポイント名〉 =
  parse 〈正則表現〉 { 〈アクション〉 }
    |   〈正則表現〉 { 〈アクション〉 }
    |   ...
and 〈エントリポイント名〉 =
  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．
- 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 ML では区切り文字として無視するため，トークンは生成せず，後続の文字列から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．続いているのは，記号に関する定義である．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 Not_found 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 を解説した本や Objective Caml マニュアルを参照すること．

```
{
let reservedWords = [
  (* Keywords in the alphabetical order *)
  ("else", Parser.ELSE);
  ("false", Parser.FALSE);
  ("if", Parser.IF);
  ("then", Parser.THEN);
  ("true", Parser.TRUE);
]
}

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 }
| ";;" { Parser.SEMISEMI }
| "+" { Parser.PLUS }
| "*" { Parser.MULT }
| "<" { Parser.LT }

| ['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: ML[1] インタプリタ: `lexer.mll`

3.3.3 environment.ml, eval.ml: 環境の実現，解釈部

式の表す値　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 (定義される言語の) 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]．今回の実験を行う範囲で実装する機能の範囲では，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．例えば，C 言語では，変数は，値そのものにつけられた名前ではなく，値が格納された箱につけられた名前と考えられるため，denoted value は expressed value への参照と考えるのが自然になる．ML[1] の場合，式の表しうる集合 Expressed Value は

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

と与えられる．+ は直和を示している．

このことを表現した Objective Caml の型宣言を以下に示す．

```
(* Expressed values *)
type exval =
    IntV of int
  | BoolV of bool
and dnval = exval
```

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 の束縛関係の組から，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．

ここでは変数と denoted value の束縛を表現できれば充分なのだが，後半の型推論においても，変数に割当てられた型を表現するために同様の構造を用いるので，汎用性を考えて設計しておく．

環境の型は，変数に関連付けられる情報 (ここでは denoted value) の型を 'a として，'a t とする．環境を操作する値や関数の型，例外を示す．(environment.mli の内容である．)

```
type 'a t
exception Not_bound
val empty : 'a t
val extend : Syntax.id -> 'a -> 'a t -> 'a t
val lookup : Syntax.id -> 'a t -> 'a
val map : ('a -> 'b) -> 'a t -> 'b t
val fold_right : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

最初の値 empty_env は，何の変数も束縛されていない，空の環境である．次の extend_env は，環境に新しい束縛をひとつ付け加えるための関数で，extend_env id dnval env で，環

[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 'a t = (Syntax.id * 'a) list

exception Not_bound

let empty = []
let extend x v env = (x,v)::env

let rec lookup x env =
  try List.assoc x env with Not_found -> raise Not_bound

let rec map f = function
    [] -> []
  | (id, v)::rest -> (id, f v) :: map f rest

let rec fold_right f env a =
  match env with
      [] -> a
    | (_, v)::rest -> f v (fold_right f rest a)
```

図 4: ML[1] インタプリタ: 環境の実装 (environment.ml)

境 env に対して，変数 id を denoted value dnval に束縛したような新しい環境を表す．関数 lookup は，環境から変数が束縛された値を取り出すもので，lookup 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 Not_bound が発生する．

また，関数 map は，map f env で，各変数が束縛された値に f を適用したような新しい環境を返す．fold_right は環境中の値を新しいものから順に左から並べたようなリストに対して fold_right を行なう．これらは，後に型推論の実装などで使われる．

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，単なる連想リストである．ただし，environment.mli では 'a t の定義を示していないので，環境を使う側は，その事実を活用することはできない．

以下は後述する main.ml に記述されている，プログラム実行開始時の環境 (大域環境) の定義である．

```
let initial_env =
  Environment.extend "i" (IntV 1)
    (Environment.extend "v" (IntV 5)
       (Environment.extend "x" (IntV 10) Environment.empty))
```

i，v，x が，それぞれ 1，5，10 に束縛されていることを表している．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．

解釈部の主要部分　以上の準備をすると，残りは，二項演算子によるプリミティブ演算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

```
let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
    Plus, IntV i1, IntV i2 -> IntV (i1 + i2)
  | Plus, _, _ -> err ("Both arguments must be integer: +")
  | Mult, IntV i1, IntV i2 -> IntV (i1 * i2)
  | Mult, _, _ -> err ("Both arguments must be integer: *")
  | Lt, IntV i1, IntV i2 -> BoolV (i1 < i2)
  | Lt, _, _ -> err ("Both arguments must be integer: <")

let rec eval_exp env = function
    Var x ->
      (try Environment.lookup x env with
        Environment.Not_bound -> err ("Variable not bound: " ^ x))
  | ILit i -> IntV i
  | BLit b -> BoolV b
  | BinOp (op, exp1, exp2) ->
      let arg1 = eval_exp env exp1 in
      let arg2 = eval_exp env exp2 in
      apply_prim op arg1 arg2
  | IfExp (exp1, exp2, exp3) ->
      let test = eval_exp env exp1 in
        (match test with
            BoolV true -> eval_exp env exp2
          | BoolV false -> eval_exp env exp3
          | _ -> err ("Test expression must be boolean: if"))

let eval_decl env = function
    Exp e -> let v = eval_exp env e in ("-", env, v)
```

図 5: ML[1] インタプリタ: 評価部の実装 (eval.ml) の抜粋

して図 5 のように定義する．eval_exp では，整数・真偽値リテラル (ILit, BLit) はそのまま値に，変数は lookup 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 apply_prim を呼んでいる．apply_prim は与えられた二項演算子の種類にしたがって，対応する Objective Caml の演算をしている．if 式の場合には，まず条件式のみを評価して，その値によって then 節/else 節の式を評価している．関数 err は，エラー時に例外を発生させるための関数である (eval.ml 参照のこと).

eval_decl は ML[1] の範囲では単に式の値を返すだけのものでよいのだが，後に，let 宣言などを処理する時のことを考えて，新たに宣言された変数名 (ここではダミーの "-") と宣言によって拡張された環境を返す設計になっている．

3.3.4 main.ml

メインプログラム main.ml を図 6 に示す．関数 read_eval_print で，

1. 入力文字列の読み込み・構文解析

```
open Syntax
open Eval

let rec read_eval_print env =
  print_string "# ";
  flush stdout;
  let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
  let (id, newenv, v) = eval_decl env decl in
    Printf.printf "val %s = " id;
    pp_val v;
    print_newline();
    read_eval_print newenv

let initial_env =
  Environment.extend "i" (IntV 1)
    (Environment.extend "v" (IntV 5)
       (Environment.extend "x" (IntV 10) Environment.empty))

let _ = read_eval_print initial_env
```

図 6: mini Scheme[1] インタプリタ: main.ml

2. 解釈

3. 結果の出力

処理を繰返している．まず，let decl = の右辺で字句解析部・構文解析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読み込むため Lexing.from_channel を使って作られている) をとっている．pp_val は eval.ml で定義されている，値をディスプレイに出力するための関数である．

Exercise 3.1 [**必修課題**] ML[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，

iv + iii * ii

などを試してみよ．

Exercise 3.2 [⋆] 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 3.3 [⋆] 論理値演算のための二項演算子 &&, || を追加せよ．

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．

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．

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

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．この拡張に伴なうプログラムの変更点を図 7 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 3.4 [**必修課題**] ML^2 インタプリタを作成し，テストせよ．

Exercise 3.5 [$\star$] Objective Caml 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 3.6 [$\star$] 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，そのファイルの内容を解釈し，結果をディスプレイに出力するように変更せよ．また，コメントを無視するよう実装せよ．

Exercise 3.7 [$\star\star$] 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 :
     Expr SEMISEMI { Exp $1 }
   | LET ID EQ Expr SEMISEMI { Decl ($2, $4) }

Expr :
     IfExpr { $1 }
   | LetExpr { $1 }
   | LTExpr { $1 }

LetExpr :
     LET ID EQ Expr IN Expr { LetExp ($2, $4, $6) }
```

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) ->
       let value = eval_exp env exp1 in
         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 7: 局所定義

3.5 ML^3 — 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関数 (二項演算子) しか提供されておらず，ML プログラマが (プリミティブを組み合わせて) 新しい関数を定義することはできなかった．ML^3 では，fun 式による関数抽象と，関数適用を提供する．

3.5.1 関数式と適用式

まずは，ML^3 の式の文法を示す．

$$\begin{aligned}\langle \text{式} \rangle \ ::= \ & \ldots \\ | \ & \ \mathsf{fun}\ \langle \text{識別子} \rangle \rightarrow \langle \text{式} \rangle \\ | \ & \ \langle \text{式}_1 \rangle \langle \text{式}_2 \rangle\end{aligned}$$

構文に関するインタプリタ・プログラムは，図 8 に示す．適用式は左結合で，他の全ての演算子よりも結合が強いとする．

3.5.2 関数閉包と適用式の評価

さて，Objective Caml と同様，ML^3 においても，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．

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

さて，関数値をどのように表現するかを考える．fun 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，本体の式の情報が必要である．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 in
let addx = fun y → x + y in
addx 4
```

x の静的束縛を実現するためには，この addx の適用を行う際には本体中の x が 2 であることを記録しておかなければならない．というわけで，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

syntax.ml:

```
type exp =
   ...
  | FunExp of id * exp
  | AppExp of exp * exp
```

parser.mly:

```
%token RARROW FUN

Expr :
   ...
   | FunExpr { $1 }

MExpr :
    MExpr MULT AppExpr { BinOp (Mult, $1, $3) }
  | AppExpr { $1 }

AppExpr :
    AppExpr AExpr { AppExp ($1, $2) }
  | AExpr { $1 }
```

lexer.mll:

```
let reservedWords = [
   ...
  ("fun", Parser.FUN);
   ...
]
...
| "=" { Parser.EQ }
| "->" { Parser.RARROW }
```

図 8: 関数と適用 (1)

eval.ml:

```
type exval =
    IntV of int
  | BoolV of bool
  | ProcV of id * exp * dnval Environment.t
and dnval = exval

let rec eval_exp env = function
   ...
  | FunExp (id, exp) -> ProcV (id, exp, env)
  | AppExp (exp1, exp2) ->
      let funval = eval_exp env exp1 in
      let arg = eval_exp env exp2 in
       (match funval with
            ProcV (id, body, env') ->
              let newenv = Environment.extend id arg env' in
                eval_exp newenv body
          | _ -> err ("Non-function value is applied"))
```

図 9: 関数と適用 (3)

3. 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，これを関数値として用いる．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，fun 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．

以上を踏まえた `eval.ml` への主な変更点は図 9 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，`exval` と `dnval` の定義が (相互) 再帰的になる．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 `FunExp` 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 3.8 [**必修課題**] ML^3 インタプリタを作成し，高階関数が正しく動作するかなどを含めてテストせよ．

Exercise 3.9 [$\star\star$] Objective Caml での「(中置演算子) 」記法をサポートし，プリミティブ演算を通常の関数と同様に扱えるようにせよ．例えば

```
let threetimes = fun f → fun x → f (f x x) (f x x) in
  threetimes (+) 5
```

は，20 を出力する．

Exercise 3.10 [⋆] Objective Caml の

```
fun x1 ... xn → ...
let f x1 ... xn = ...
```

といった簡略記法をサポートせよ．

Exercise 3.11 [⋆] 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 ML^3 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 = fun maker → fun x →
                 if x < 1 then 0 else 4 + maker maker (x + -1) in
let times4 = fun x → makemult makemult x in
  times4 3
```

Exercise 3.12 [⋆] 静的束縛とは対照的な概念として**動的束縛**(*dynamic binding*) がある．動的束縛の下では，関数本体は，fun 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 in
let p = fun x → x + a in
let a = 5 in
  a * p 2
```

というプログラムで，関数 p 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，動的束縛を実現せよ．

Exercise 3.13 [⋆] 動的束縛の下では，ML^4 で導入するような再帰定義を実現するための特別な仕組みや，Exercise 3.11 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 = fun n → n + 1 in
let fact = fun n → if n < 1 then 1 else n * fact (n + -1) in
  fact 5
```

3.6 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．

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

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． 再帰的定義の場合，定義される関数の閉包内の環境を，**今これから作ろうとしている束縛関係まで含んでいるもの**にしたい．これを実現するために，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，たった今拡張した環境に**更新**する，という手法である．Objective Caml で実装する際には，関数閉包の環境を更新できるように，(例えば) 参照を用いる．

図 10 が，主なプログラムの変更点である．`eval_exp` の `LetRecExp` を処理する部分は，上で述べたバックパッチをまさに実現している．

Exercise 3.14 [**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，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 3.15 [$\star\star$] 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 rec 式・宣言を拡張し，相互再帰的関数をテストせよ．

3.7 ML^5 — リスト

Exercise 3.16 [$\star\star$] 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，

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

程度の制限されたものでよい．

Exercise 3.17 [$\star$] リスト表記

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

をサポートせよ．

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) ->
          let dummyenv = ref Environment.empty in
          let newenv =
              Environment.extend id (ProcV (para, exp1, dummyenv)) env in
           dummyenv := newenv;
           eval_exp newenv exp2
```

図 10: 再帰的関数定義

Exercise 3.18 [$\star$] match 式において，リストの先頭と残りを表す変数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 3.19 [$\star\star\star$] より一般的なパターンマッチ構文を実装せよ．

Exercise 3.20 [$\star\star$] ここまで与えた構文規則では，Objective Caml 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．

4 型推論機構の実装

4.1 ML^2 のための型推論

まず，ML^2 の式に対しての型推論を考える．ML^2 では，トップレベル入力として，式だけでなく let 宣言を導入したが，ここではひとまず式についての型推論のみを考え，let 宣言については最後に扱うことにする．ML^2 の文法は (記法は多少異なるが) 以下のようであった．

$$\begin{aligned} e &::= x \mid n \mid \mathsf{true} \mid \mathsf{false} \mid e_1\ \mathbf{op}\ e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ &\mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ \mathbf{op} &::= + \mid * \mid < \end{aligned}$$

ここでは $\langle$ 式 $\rangle$ の代わりに e という記号 (メタ変数)，$\langle$ 識別子 $\rangle$ の代わりに x という記号 (メタ変数) を用いている．また，型 (メタ変数 τ) として，整数を表す **int**, 真偽値を表す **bool** を考える．

$$\tau ::= \mathbf{int} \mid \mathbf{bool}$$

4.1.1 型判断と型付け規則

さて，型推論のアルゴリズムを考える前に，そもそも「式 e が型 τ を持つ」という関係がどのような時に成立するかを正確に記述したい．例えば「式 $1+1$ は型 **int** を持つ」だろうが,「式 if 1 then $2+3$ else 4 は型 **int** を持つ」は成立しないと思われる．この,「式 e が型 τ を持つ」という判断を**型判断**(*type judgment*) と呼び，$e : \tau$ と略記する．

しかし，一般に式には変数が現れるため，例えば単に x が **int** を持つか，といわれても判断することができない．このため，変数に対しては，それが持つ型を何か仮定しないと型判断は下せないことになる．この，変数に対して仮定する型に関する情報を**型環境**(*type environment*)(メタ変数 Γ) と呼び，変数から型への部分関数で表される．これを使えば，変数に対する型判断は，例えば

$\Gamma(x) = \mathbf{int}$ の時 $x : \mathbf{int}$ である

と言える．このことを考慮に入れて，型判断は，$\Gamma \vdash e : \tau$ と記述し，

型環境 Γ の下で式 e は型 τ を持つ

と読む．$\vdash$ は数理論理学などで使われる記号で「〜という仮定の下で判断〜が導出・証明される」くらいの意味である．インタプリタが (変数を含む) 式の値を計算するために環境を使ったように，型推論器が式の型を計算するために型環境を使っているとも考えられる．

型判断は，**型付け規則**(*typing rule*) を使って導出される．型付け規則は一般に

$$\frac{\langle \text{型判断}_1 \rangle \qquad \cdots \qquad \langle \text{型判断}_n \rangle}{\langle \text{型判断} \rangle} \qquad (\langle \text{規則名} \rangle)$$

という形で書かれ，横線の上の$\langle$ 型判断 $_1\rangle$,...,$\langle$ 型判断 $_n\rangle$を規則の**前提**(*premise*)，下にある$\langle$ 型判断 $\rangle$を規則の**結論**(*conclusion*) と呼ぶ．例えば，以下は加算式の型付け規則である．

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \qquad \text{(T-Plus)}$$

この，型付け規則の直感的な意味 (読み方) は，

前提の型判断が全て導出できたならば，結論の型判断を導出してよい

ということである．型判断は，具体的な型環境，式，型に関して，導出するものなので，より厳密には，規則を使うという場合には，まず，規則に現れる，Γ, e_1 などのメタ変数を実際の型環境，式などで具体化して使わなければいけない．例えば，$\emptyset \vdash 1 : \mathbf{int}$ という型判断が既に導出されていたとしたら ($\emptyset$ は空の型環境を表す)，この規則の Γ を $\emptyset$ に，e_1, e_2 をともに，1 に具体化することによって，規則の**インスタンス，具体例**(*instance*)

$$\frac{\emptyset \vdash 1 : \mathbf{int} \qquad \emptyset \vdash 1 : \mathbf{int}}{\emptyset \vdash 1 + 1 : \mathbf{int}} \qquad \text{(T-Plus)}$$

が得られる．そこで，この具体化された規則を使うと，型判断 $\emptyset \vdash 1 + 1 : \mathbf{int}$ が導出できる．ちなみに，一般に何もないところから，ある型判断を導出するには，前提が無い型付け規則から始めることになる．

以下に，ML^2 の型付け規則を示す．

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau} \qquad \text{(T-Var)}$$

$$\frac{}{\Gamma \vdash n : \mathbf{int}} \qquad \text{(T-Int)}$$

$$\frac{(b = \texttt{true} \text{ または } b = \texttt{false})}{\Gamma \vdash b : \mathbf{bool}} \quad \text{(T-Bool)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \quad \text{(T-Plus)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 * e_2 : \mathbf{int}} \quad \text{(T-Mult)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 < e_2 : \mathbf{bool}} \quad \text{(T-Lt)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau} \quad \text{(T-If)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \quad \text{(T-Let)}$$

規則 T-Let に現れる $\Gamma, x : \tau$ は Γ に x は τ であるという情報を加えた拡張された型環境で，より厳密な定義としては，

$$dom(\Gamma, x : \tau) = dom(\Gamma) \cup \{x\}$$
$$(\Gamma, x : \tau)(y) = \begin{cases} \tau & (\text{if } x = y) \\ \Gamma(y) & (\text{otherwise}) \end{cases}$$

と書くことができる．($dom(\Gamma)$ は Γ の定義域を表す．) 規則の前提として括弧内に書かれているのは**付帯条件**(*side condition*) と呼ばれるもので，規則を使う際に成立していなければならない条件を示している．

4.1.2 型推論アルゴリズム

以上を踏まえると，型推論アルゴリズムの仕様は，以下のように考えることができる．

入力: 型環境 Γ と式 e
出力: $\Gamma \vdash e : \tau$ という型判断が導出できるような型 τ

さて，このような仕様を満たすアルゴリズムを，どのように設計したらよいのかについては，型付け規則が指針を与えてくれる．どういうことかというと，型付け規則を下から上に読むことによってアルゴリズムが得られるのである．例えば，T-Int は入力式が整数リテラルならば，型環境に関わらず，**int** を出力する，と読むことができるし，T-Plus は，部分式の型を求めて，それが両方とも **int** であった場合には **int** 型を出力すると読むことができる．

Exercise 4.1 [**必修課題**] 図 11, 図 12 に示すコードを参考にして，型推論アルゴリズムを完成させよ．(ソースファイルとして `typing.ml` を追加するので，`make depend` の実行を一度行うこと．)

4.2 ML^3 の型推論

ここでの実装のために，集合演算を実装したモジュールである `Set` (`set.ml`, `set.mli`) を使用する．これも，`http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/class/isle4/src/` からダウンロードしておくこと．(Objective Caml の標準ライブラリと名前が同じなので，ダウンロードしておかないとコンパイル時に意味不明のエラーが発生する恐れがある．)

4.2.1 関数に関する型付け規則

次に，fun 式，関数適用式

$$e ::= \cdots \mid \mathsf{fun}\ x\ \to e \mid e_1\ e_2$$

について考える．関数の型は $\tau_1 \to \tau_2$ とすると，型の定義は以下のように変更される．

$$\tau ::= \mathbf{int} \mid \mathbf{bool} \mid \tau_1 \to \tau_2$$

そして，これらの式に関して型付け規則は以下のように与えられる．

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \text{(T-App)}$$

規則 T-Abs は，関数本体 e が，引数 x が τ_1 を持つという仮定の下で τ_2 型を持つならば，$\mathsf{fun}\ x \to e$ は $\tau_1 \to \tau_2$ 型である，と読める．また，規則 T-App は，e_1 の型が，そもそも関数型であり，かつ，その引数型と e_2 の型が一致している場合に，適用式全体に e_1 の返り値型がつくことを示している．

この規則について，前節と同様に，規則を下から上に読むことでアルゴリズムを与えようとすると T-Abs で問題が生じる．というのも，e の型を調べようとする時に x の型として何を与えてよいかわからないためである．簡単な例として，$\mathsf{fun\ x} \to \mathsf{x} + 1$ という式を考えてみよう．これは，**int** → **int** 型の関数であることは「一目で」わかるので，一見，x の型を **int** として推論を続ければよさそうだが，問題は，本体式である $\mathsf{x} + 1$ を見るまえには，x の型が **int** であることがわからない，というところにある．

Makefile:

```
OBJS=syntax.cmo parser.cmo lexer.cmo \
   environment.cmo typing.cmo eval.cmo main.cmo
```

syntax.ml:

```
type ty =
    TyInt
  | TyBool

let pp_ty = function
      TyInt -> print_string "int"
    | TyBool -> print_string "bool"
```

main.ml:

```
open Typing

let rec read_eval_print env tyenv =
   print_string "# ";
   flush stdout;
   let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
   let ty = ty_decl tyenv decl in
   let (id, newenv, v) = eval_decl env decl in
     Printf.printf "val %s : " id;
     pp_ty ty;
     print_string " = ";
     pp_val v;
     print_newline();
     read_eval_print newenv tyenv

let initial_tyenv =
   Environment.extend "i" TyInt
     (Environment.extend "v" TyInt
       (Environment.extend "x" TyInt  Environment.empty))

let _ = read_eval_print initial_env initial_tyenv
```

図 11: ML^2 型推論の実装 (1)

typing.ml:

```
    open Syntax

    exception Error of string

    let err s = raise (Error s)

    (* Type Environment *)
    type tyenv = ty Environment.t

    let ty_prim op ty1 ty2 = match op with
        Plus -> (match ty1, ty2 with
                     TyInt, TyInt -> TyInt
                   | _ -> err ("Argument must be of integer: +"))
        ...
      | Cons -> err "Not Implemented!"

    let rec ty_exp tyenv = function
        Var x ->
          (try Environment.lookup x tyenv with
              Environment.Not_bound -> err ("variable not bound: " ^ x))
      | ILit _ -> TyInt
      | BLit _ -> TyBool
      | BinOp (op, exp1, exp2) ->
          let tyarg1 = ty_exp tyenv exp1 in
          let tyarg2 = ty_exp tyenv exp2 in
            ty_prim op tyarg1 tyarg2
      | IfExp (exp1, exp2, exp3) ->
          ...
      | LetExp (id, exp1, exp2) ->
          ...
      | _ -> err ("Not Implemented!")

    let ty_decl tyenv = function
        Exp e -> ty_exp tyenv e
      | _ -> err ("Not Implemented!")
```

図 12: ML^2 型推論の実装 (2)

4.2.2 型変数，型代入と型推論アルゴリズムの仕様

この問題を解決するために,「今のところ正体がわからない未知の型」を表す**型変数**(*type variable*) を導入する[5].

$$\tau ::= \alpha \mid \mathbf{int} \mid \mathbf{bool} \mid \tau_1 \to \tau_2$$

そして，型推論アルゴリズムの出力として，入力 (正確には型環境中) に現れる型変数の「正体が何か」を返すことにする．上の例だと，とりあえず x の型は α などと置いて，型推論を続ける．推論の結果，$\mathsf{x}+1$ の型は **int** である，という情報に加え $\alpha = \mathbf{int}$ という「型推論の結果 α は **int** であることが判明しました」という情報が返ってくることになる．最終的に T-Abs より，全体の型は $\alpha \to \mathbf{int}$，つまり，$\mathbf{int} \to \mathbf{int}$ であることがわかる．また，$\mathsf{fun\ x} \to \mathsf{fun\ y} \to \mathsf{x\ y}$ のような式を考えると，以下のような手順で型推論がすすむ.

1. x の型を α と置いて，本体，つまり $\mathsf{fun\ y} \to \mathsf{x\ y}$ の型推論を行う.
2. y の型を別の型変数 β と置いて，本体，つまり $\mathsf{x\ y}$ の型推論を行う.
3. $\mathsf{x\ y}$ の型推論の結果，この式の型が (さらに別の型変数) γ であり，$\alpha = \beta \to \gamma$ であることが判明する.

さらに詳しい，型推論の処理については後述するが，ここで大事なことは，とりあえず未知の型として用意した型変数の正体が，推論の過程で徐々に明らかになっていくことである.

ここまで述べたことを実装したのが図 13 である．ここでは型変数は整数を使って表現している．関数 `fresh_tyvar` は `fresh_tyvar ()` とすることで，新しい未使用の型変数を生成する．これは，T-Abs のように未知の型を「とりあえず α とおく」際に使われる.

上述の型変数とその正体の対応関係を，**型代入**(*type substitution*) と呼ぶ．型代入 (メタ変数として $\mathcal{S}$ を使用する．) は，型変数から型への (有限) 写像として表される．以下では，$\mathcal{S}\tau$ で τ 中の型変数を $\mathcal{S}$ を使って置き換えたような型，$\mathcal{S}\Gamma$ で，型環境中の全ての型に $\mathcal{S}$ を適用したような型環境を表す．$\mathcal{S}\tau$，$\mathcal{S}\Gamma$ はより厳密には以下のように定義される.

$$\begin{aligned}
\mathcal{S}\alpha &= \begin{cases} \mathcal{S}(\alpha) & \text{if } \alpha \in dom(\mathcal{S}) \\ \alpha & \text{otherwise} \end{cases} \\
\mathcal{S}\mathbf{int} &= \mathbf{int} \\
\mathcal{S}\mathbf{bool} &= \mathbf{bool} \\
\mathcal{S}(\tau_1 \to \tau_2) &= \mathcal{S}\tau_1 \to \mathcal{S}\tau_2
\end{aligned}$$

$$\begin{aligned}
dom(\mathcal{S}\Gamma) &= dom(\Gamma) \\
(\mathcal{S}\Gamma)(x) &= \mathcal{S}(\Gamma(x))
\end{aligned}$$

型代入を使って，新しい型推論アルゴリズムの仕様は以下のように与えられる.

[5] これまで型変数は「何の型にでも置き換えてよい」ものとして説明してきたが，ここでの型変数は「特定の型を表す」記号であり，代数方程式における変数に近い.「任意の型で置き換え可能」な型変数は次節で再登場する.

入力: 型環境 Γ と式 e
出力: $\mathcal{S}\Gamma \vdash e : \tau$ を結論とする判断が存在するような型 τ と代入 $\mathcal{S}$

Exercise 4.2 [**必修課題**] 図 13 中の `pp_ty`, `freevar_ty` を完成させよ. `freevar_ty` は, 与えられた型中の型変数の集合を返す関数で, 型は

```
val freevar_ty : ty -> tyvar Set.t
```

とする. 型 `'a Set.t` は (実験 WWW ページにある) `set.mli` で定義されている `'a` を要素とする集合を表す型である.

Exercise 4.3 [**必修課題**] 型代入に関する以下の型, 関数を `typing.ml` 中に実装せよ.

```
type subst

val empty_subst : subst
val atomic_subst : tyvar -> ty -> subst
val subst_type : subst -> ty -> ty
val subst_tenv : subst -> tyenv -> tyenv
val (@@) : subst -> subst -> subst
```

但し, `empty_subst` は空の型代入であり, `atomic_subst id ty` は型変数 `id` を `ty` に写す (だけの) 型代入を返す. `subst_type subst ty` や `subst_tenv subst tyenv` は型代入 `subst` を型 `ty` や型環境 `tyenv` に適用した結果を返す. 例えば,

```
let alpha = fresh_tyvar () in
subst_type (atomic_subst alpha TyInt) (TyFun (alpha, TyBool))
```

の値は `TyFun (TyInt, TyBool)` になり,

```
let alpha = fresh_tyvar () in
let beta = fresh_tyvar () in
subst_type (atomic_subst alpha TyBool)
  (subst_type (atomic_subst beta (TyFun (TyVar alpha, TyInt))) (TyVar beta))
```

の値は `TyFun (TyBool, TyInt)` になる. `subst1 @@ subst2` は代入の合成を示しており,

```
subst_type (subst1 @@ subst2) ty
```

と

```
subst_type subst1 (subst_type subst2 ty)
```

の値が等しくならなければいけない. 例えば,

```
let alpha = fresh_tyvar () in
let beta = fresh_tyvar () in
subst_type (atomic_subst alpha TyBool
            @@ atomic_subst beta (TyFun (TyVar alpha, TyInt))) (TyVar beta)
```

の値は `TyFun (TyBool, TyInt)` になる.

Makefile:

```
OBJS= set.cmo syntax.cmo parser.cmo lexer.cmo \
   environment.cmo eval.cmo main.cmo
```

syntax.ml:

```
...
type tyvar = int

type ty =
    TyInt
  | TyBool
  | TyVar of tyvar
  | TyFun of ty * ty

(* pretty printing *)
let pp_ty = ...

let fresh_tyvar =
  let counter = ref 0 in
  let body () =
    let v = !counter in
      counter := v + 1; v
  in body

let rec freevar_ty ty = ... (*  ty -> tyvar Set.t *)
```

図 13: ML^3 型推論の実装 (1)

4.2.3 単一化

型付け規則を眺めてみると，T-If や T-Plus などの規則は「条件式の型は **bool** でなくてはならない」「then 節と else 節の式の型は一致していなければならない」「引数の型は **int** でなくてはならない」という制約を課していることがわかる．ML^2 に対する型推論では，`... = TyInt` といった型 (定義される言語の型を表現した値) の比較を行うことで，こういった制約が満たされていることを検査していた．

しかし，部分式の型として，型変数が含まれる可能性がでてくるため，そういった制約の検査には，このような単純比較は不十分である．例えば，$\mathsf{fun\ x} \rightarrow 1 + \mathsf{x}$ という式の型推論過程を考えてみる．上で述べたように，まず，x の型を型変数 α とおいて，$1 + \mathsf{x}$ の型推論をすることになる．まず，部分式の型推論をするわけだが，この場合，それぞれの部分式 $1, \mathsf{x}$ の型は，**int** と α となる．しかし，後者の型が **int** ではない (単純比較に失敗する) からといって，この式に型がつかないわけではない．ここでわかることは「未知だった α は実は **int** である」ということである．つまり，型に関する制約を課している部分で未知の型の正体が求まるのである．この例の場合の処理は単純だが，T-If で then 節と else 節の式の型が一致することを検査するためには，より一般的な，

与えられたふたつの型 τ_1, τ_2 に対して，$\mathcal{S}\tau_1 = \mathcal{S}\tau_2$ なる $\mathcal{S}$ を求める

という問題を解かなければいけない．このような問題は**単一化**(*unification*) 問題と呼ばれ，型推論だけではなく，計算機による自動証明などにおける基本的な問題として知られている．

例えば，α, **int** は $\mathcal{S}(\alpha) = \mathbf{int}$ なる型代入 $\mathcal{S}$ により単一化できる．また，$\alpha \rightarrow \mathbf{bool}$ と $(\mathbf{int} \rightarrow \beta) \rightarrow \beta$ は $\mathcal{S}(\alpha) = \mathbf{int} \rightarrow \mathbf{bool}$ かつ $\mathcal{S}(\beta) = \mathbf{bool}$ なる $\mathcal{S}$ により単一化できる．

単一化問題は，対象 (ここでは型) の構造や変数の動く範囲によっては，非常に難しくなるが，ここでは，型が単純な木構造を持ち，型代入も単に型変数に型を割当てるだけのもの (**一階の単一化**(*first-order unification*) と呼ばれる) なので，解である型代入を求めるアルゴリズムが存在する．(しかも，求まる型代入がある意味で「最も良い」解であることがわかっている．) ふたつの型を入力とし型代入を返す，単一化のアルゴリズムは，以下のように与えられる．

$$
\begin{array}{lcl}
\mathcal{U}(\tau, \tau) & = & \emptyset \\
\mathcal{U}(\alpha, \tau) & = & \begin{cases} [\tau \mapsto \alpha] & (\alpha \notin FTV(\tau)) \\ \mathbf{error} & (\text{その他}) \end{cases} \\
\mathcal{U}(\tau, \alpha) & = & \begin{cases} [\tau \mapsto \alpha] & (\alpha \notin FTV(\tau)) \\ \mathbf{error} & (\text{その他}) \end{cases} \\
\mathcal{U}(\tau_{11} \rightarrow \tau_{12}, \tau_{21} \rightarrow \tau_{22}) & = & \mathcal{U}(\mathcal{S}\tau_{12}, \mathcal{S}\tau_{22}) \quad (\mathcal{S} = \mathcal{U}(\tau_{11}, \tau_{21})) \\
\mathcal{U}(\tau_1, \tau_2) & = & \mathbf{error} \quad (\text{その他の場合})
\end{array}
$$

$\emptyset$ は空の型代入を表し，$[\alpha \mapsto \tau]$ は α を τ に写すだけの型代入である．また $FTV(\tau)$ は τ 中に現れる型変数の集合である．関数型同士の単一化は，定義域の型・値域の型をそれぞれ単一化するのだが，一方の単一化の結果得られた型代入をもう一方に適用してから単一化を行うことに注意すること．

Exercise 4.4 [**必修課題**] 単一化アルゴリズムにおいて，$\alpha \notin FTV(\tau)$ という条件はなぜ必要か考察せよ．

4.2.4 ML^3 型推論アルゴリズム

以上を総合すると，ML^3 のための型推論アルゴリズムが得られる．例えば，e_1+e_2 式に対する型推論は，T-PLUS 規則を下から上に読んで，

1. Γ, e_1 を入力として型推論を行い，$\mathcal{S}_1, \tau_1$ を得る．
2. $\mathcal{S}_1\Gamma, e_2$ を入力として型推論を行い， $\mathcal{S}_2, \tau_2$ を得る．
3. $\mathcal{S}_2\tau_1$ と **int** を単一化し，型代入 $\mathcal{S}_3$ を得る．
4. $\mathcal{S}_3\tau_2$ と **int** を単一化し，型代入 $\mathcal{S}_4$ を得る．
5. $\mathcal{S}_4 \circ \mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1$ と **int** を出力として返す．

となる．部分式の型推論や単一化で得られた型代入は部分的な解を与えているので，残りの処理を行う前に適用していることに注意せよ．(関数型に対する単一化の処理と同様である．)

Exercise 4.5 [**必修課題**] 他の型付け規則に関しても同様に型推論の手続きを考え，図 14 を参考にして，型推論アルゴリズムの実装を完成させよ．

Exercise 4.6 [$\star\star$] 再帰的定義のための let rec 式の型付け規則は以下のように与えられる．

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e_1 : \tau_2 \qquad \Gamma, f : \tau_1 \to \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let\ rec}\ f = \mathsf{fun}\ x\ \to e_1\ \mathsf{in}\ e_2 : \tau} \quad \text{(T-LETREC)}$$

型推論アルゴリズムが let rec 式を扱えるように拡張せよ．

Exercise 4.7 [$\star\star$] 以下は，リスト操作に関する式の型付け規則である．リストには要素の型を τ として τ **list** という型を与える．

$$\frac{}{\Gamma \vdash [] : \tau\ \mathbf{list}} \quad \text{(T-NIL)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \mathbf{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \mathbf{list}} \quad \text{(T-CONS)}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \mathbf{list} \qquad \Gamma \vdash e_2 : \tau' \qquad \Gamma, x : \tau, y : \tau\ \mathbf{list} \vdash e_3 : \tau'}{\Gamma \vdash \mathsf{match}\ e_1\ \mathsf{with}\ [] \to e_2 \mid x :: y \to e_3 : \tau'} \quad \text{(T-MATCH)}$$

型推論アルゴリズムがこれらの式を扱えるように拡張せよ．

typing.ml:

```
let rec unify ty1 ty2 = ...

let ty_prim op ty1 ty2 = match op with
    Plus ->
      let subst1 = unify ty1 TyInt in
      let subst2 = unify (subst_type subst1 ty2) TyInt in
       (subst2 @@ subst1, TyInt)
  | ...

let rec ty_exp tyenv = function
    Var x ->
     (try (empty_subst, Environment.lookup x tyenv) with
         Environment.Not_bound -> err ("variable not bound: " ^ x))
  | ILit _ -> (empty_subst, TyInt)
  | BLit _ -> (empty_subst, TyBool)
  | BinOp (op, exp1, exp2) ->
      let (subst1, ty1) = ty_exp tyenv exp1 in
      let (subst2, ty2) = ty_exp (subst_tenv subst1 tyenv) exp2 in
      let (subst3, ty) = ty_prim op (subst_type subst2 ty1) ty2 in
        (subst3 @@ subst2 @@ subst1, ty)
  | IfExp (exp1, exp2, exp3) -> ...
  | LetExp (id, exp1, exp2) -> ...
  | FunExp (id, exp) ->
      let domty = TyVar (fresh_tyvar ()) in
      let subst, ranty =
       ty_exp (Environment.extend id domty tyenv) exp in
       (subst, TyFun (subst_type subst domty, ranty))
  | AppExp (exp1, exp2) -> ...
  | _ -> Error.typing ("Not Implemented!")
```

図 14: ML^3 型推論の実装 (2)

4.3 多相的 let の型推論

前節までの実装で実現される型 (システム) は単相的であり，ひとつの変数をあたかも複数の型を持つように扱えない．例えば，

```
let f = fun x → x  in
if f true then f 2 else 3;;
```

のようなプログラムは，f が，if の条件部では **bool** → **bool** として，また，then 節では **int** → **int** として使われているため， 型推論に失敗してしまう．本節では，上記のプログラムなどを受理するよう let 多相を実装する．

4.3.1 多相性と型スキーム

Objective Caml で `let f = fun x -> x;;` とすると，その型は `'a -> 'a` であると表示される．しかし，ここで現れる型変数 `'a` は，後でその正体が判明する (今のところは) 未知の型を表しているわけではなく,「どんな型にでも置き換えてよい」ことを示すための，いわば「穴ボコ」につけた名前である．そのために，`'a` を `int` で置き換えて `int->int` として扱って整数に適用としたり，`'a` を置き換えて `bool->bool` として真偽値に適用したりすることができる．このような型変数の役割の違いを表すために，任意の型に置き換えてよい変数は，型の前に $\forall\alpha.$ をつけて未知の型を表す変数と区別する．このような表現を**型スキーム**(*type scheme*) と呼ぶ．例えば $\forall\alpha.\alpha \to \alpha$ は型スキームである．型は $\forall\alpha.$ がひとつもついていない型スキームと考えられるが，型スキームは一般に型ではない．例えば，$(\forall\alpha.\alpha) \to (\forall\alpha.\alpha)$ のような表現は型とは見做されない．(型スキームを型として扱えるようなプログラミング言語も存在するが，素朴に同一視すると型推論ができなくなってしまう．) 型スキームは σ で表す．型と型スキームの定義は以下の通りである．

$$\begin{aligned} \tau &::= \alpha \mid \mathbf{int} \mid \mathbf{bool} \mid \tau_1 \to \tau_2 \\ \sigma &::= \tau \mid \forall\alpha.\sigma \end{aligned}$$

型スキーム中，$\forall$ のついている型変数を束縛されているといい，束縛されていない型変数 (これらは未知の型を表す型変数である) を自由である，という． 例えば $\forall\alpha.\alpha \to \alpha \to \beta$ において，α は束縛されており，β は自由である．図 15 上半分に型スキームの実装上の定義を示す．関数 `freevar_tysc` は型スキームからその中の自由な型変数 (の集合) を求める関数である．

次に，型スキームをもとに型付け規則がどのようになるか考えてみる．まず，一般に let で定義された変数は型スキームを持つので，型環境は型から型スキームへの写像となる．そして，まず変数の型付け規則は以下のように書ける．

$$\frac{(\Gamma(x) = \forall\alpha_1, \ldots, \alpha_n.\tau)}{\Gamma \vdash x : [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]\tau} \qquad \text{(T-PolyVar)}$$

この規則は，変数の型スキーム中の $\forall$ のついた型変数は任意の型に置き換えてよいことを，型代入 $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ を使って表現している．例えば，$\Gamma(\mathsf{f}) = \forall\alpha.\alpha \to \alpha$ とすると，

$$\Gamma \vdash \mathsf{f} : \mathbf{int} \to \mathbf{int}$$

や

$$\Gamma \vdash \mathsf{f} : (\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int})$$

といった型判断を導出することができる．さて，let に関しては，大まかには以下のような規則になる．

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \text{(T-PolyLet')}$$

これは，e_1 の型から型スキームを作って，それを使って e_2 の型付けをすればいいことを示している．さて，残る問題は $\alpha_1, \ldots, \alpha_n$ としてどんな型変数を選べばよいかである．もちろん，τ_1 に現れる型変数に関して $\forall$ をつけて,「未知の型」から「任意の型」に役割変更をするのだが，どんな型変数でも変更してよいわけではない．役割変更してよいものは Γ に自由に出現しないものである．Γ 中に (自由に) 現れる型変数は，その後の型推論の過程で正体がわかって特定の型に置き換えられる可能性があるので，任意におきかえられるものとみなしてはまずいのである．というわけで，正しい型付け規則は，付帯条件をつけて，

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau_1 \vdash e_2 : \tau_2 \qquad (\alpha_1, \ldots, \alpha_n \text{ は } \tau_1 \text{ に自由に出現する型変数で } \Gamma \text{ には自由に出現しない})}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \text{(T-PolyLet)}$$

となる．図 16 の関数 `closure` が， τ_1 と Γ から条件を満たす型スキームを計算する関数である．(`ids` が上での $\alpha_1, \ldots, \alpha_n$ に 対応する．)

4.3.2 型スキームに対する型代入

型推論の過程において (型環境中の) 型スキームに対して型代入を作用させることがある．この際，自由な型変数と束縛された型変数をともに `tyvar` 型の値 (実際は整数) で表現しているために，型スキームへの代入の定義は多少気をつける必要がある．というのは，置き換えた型中の自由な型変数と，束縛されている型変数が同じ名前で表現されている可能性があるためである．

例えば，型スキーム $\forall\alpha.\alpha \to \beta$ に $\mathcal{S}(\beta) = \alpha \to \mathbf{int}$ であるような型代入を作用させることを考える．この代入は，β を未知の型を表す α を使った型で置き換えることを示している．しかし，素朴に型スキーム中の β を置き換えると，$\forall\alpha.\alpha \to \alpha \to \mathbf{int}$ という型スキームが得られてしまう．この型スキームでは，代入の前は，未知の型を表す型変数であった，二番目

の α までが任意に置き換えられる型変数になってしまっている．このように，代入によって型変数の役割が変化してしまうのはまずいので避けなければいけない．

このような変数の衝突問題を避けるための，ここで取る解決 (回避) 策は束縛変数の名前替え，という手法である[6]．

これは，例えば $\forall\alpha.\alpha \to \alpha$ と $\forall\beta.\beta \to \beta$ が (文字列としての見かけは違っても) 意味的には同じ型スキームを表している[7]ことを利用する．つまり，代入が起こる前に，新しい型変数を使って一斉に束縛変数の名前を替えてしまって衝突が起こらないようにするのである．上の例ならば，まず，$\forall\alpha.\alpha \to \beta$ を $\forall\gamma.\gamma \to \beta$ として，その後に β を $\alpha \to \mathbf{int}$ で置き換え，$\forall\gamma.\gamma \to \alpha \to \mathbf{int}$ を得ることになる．

このような変数の名前替えを伴う代入操作の実装を図 16 に示す．`rename_tysc`, `subst_tysc` がそれぞれ型スキームの名前替え，代入のための関数である．

4.3.3 型推論アルゴリズム概要

ここまでのところが理解できれば，実は型推論アルゴリズム自体に関して影響を受けるところは少ない．実際に大きな影響をうけるのは，変数の処理と let の処理である．変数の処理に関しては，型変数に代入する型 (型付け規則中の $\tau_1, \ldots, \tau_n$) は未知なので，新しい型変数を用意してそれらを代入することになる．(図 16 のコードでは，名前替えの関数を利用してそれを実現している．)

Exercise 4.8 [⋆⋆] 図 15, 16 を参考にして，多相的 let 式・宣言ともに扱える型推論アルゴリズムの実装を完成させよ．

Exercise 4.9 [⋆] 以下の型付け規則を参考にして，再帰関数が多相的に扱えるように，型推論機能を拡張せよ．

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e_1 : \tau_2 \qquad \Gamma, f : \forall\alpha_1, \ldots, \alpha_n.\tau_1 \to \tau_2 \vdash e_2 : \tau \quad (\alpha_1, \ldots, \alpha_n \text{ は } \tau_1 \text{ もしくは } \tau_2 \text{ に自由に出現する型変数で } \Gamma \text{ には自由に出現しない})}{\Gamma \vdash \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e_1\ \mathsf{in}\ e_2 : \tau}$$

(T-PolyLetRec)

Exercise 4.10 [⋆⋆⋆] Objective Caml では，「: ⟨型⟩」という形式で，式や宣言された変数の型を指定することができる．この機能を扱えるように処理系を拡張せよ．

Exercise 4.11 [⋆⋆⋆] 型推論時のエラー処理を，プログラマにエラー箇所がわかりやすくなるように改善せよ．

---

[6] 他にもいろいろな回避策が考えられる．「計算と論理」の講義で関連した問題に詳しく触れられる (かもしれない)．

[7] 関数などの仮引数の名前を使われている場所といっしょに変えても同じ関数を表していることと同様の現象と考えられる．

syntax.ml

```
    (* type scheme *)
    type tysc = TyScheme of tyvar list * ty

    let tysc_of_ty ty = TyScheme ([], ty)

    let freevar_tysc tysc = ...
```

main.ml

```
    ...
    let rec read_eval_print env tyenv =
      print_string "# ";
      flush stdout;
      let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
      let (newtyenv, ty) = ty_decl tyenv decl in
      let (id, newenv, v) = eval_decl env decl in
        Printf.printf "val %s : " id;
        pp_ty ty;
        print_string " = ";
        pp_val v;
        print_newline();
        read_eval_print newenv newtyenv
```

図 15: 多相的 let のための型推論の実装 (1)

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

[3] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.09: Documentation and user's manual*, 2004. `http://caml.inria.fr/ocaml/htmlman/index.html`.

[4] 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.

typing.ml

```
type tyenv = tysc Environment.t

let rec freevar_tyenv tyenv = ...

let closure ty tyenv =
  let ids = Set.diff (freevar_ty ty) (freevar_tyenv tyenv) in
    TyScheme (Set.to_list ids, ty)

...

let rec subst_type subst = ...

let rename_tysc tysc =
   let TyScheme (ids, ty) = tysc in
   let newids = List.map (fun _ -> fresh_tyvar()) ids in
   let subst_list = List.map2
            (fun id newid -> atomic_subst id (TyVar newid))
       ids newids in
   let subst = List.fold_left (fun s1 s2 -> s1 @@ s2)
                        empty_subst subst_list in
     TyScheme (newids, subst_type subst ty)

let subst_tysc subst tysc =
  let TyScheme (newids, ty') = rename_tysc tysc in
    TyScheme (newids, subst_type subst ty')

let subst_tyenv subst tenv = ...

...

let rec ty_exp tyenv = function
     Var x ->
       (try
         let TyScheme (_, renamed_ty) =
           rename_tysc (Environment.lookup x tyenv) in
           (empty_subst, renamed_ty)
        with Environment.Not_bound -> err ("variable not bound: " ^ x))
   | ...

let ty_decl tyenv = function
    Exp e -> let (_, ty) = ty_exp tyenv e in (tyenv, ty)
  | Decl (id, e) -> ...
```

図 16: 多相的 let のための型推論の実装 (2)