「プログラミング言語処理系」と「計算機科学実験及演習３（ソフトウェア）」のテキスト

末永 幸平　　　五十嵐 淳

目 次

第1章　はじめに

重要：このテキストは必要に応じてアップデートされる．アップデートしたテキストは *GitHub* 本講義のリポジトリにアップロードする．

本書は工学部専門科目「プログラミング言語処理系」と「計算機科学実験及演習3（ソフトウェア）」のテキストである．プログラミング言語の設計と実装に関わるトピックをカバーしている．

本書では (1) 工学部専門科目「プログラミング言語」と (2) 「言語・オートマトン」の内容を既知とする．実装課題に取り組む場合は特に (3) git の基本的な操作法と (4) OCaml の知識とある程度の実装力が必要である．本講義でも OCaml の復習を少しやる予定であるが，あまり時間をかけることはできないので，以下の問題が解ける程度になるまで各自自習されたい．五十嵐淳による OCaml 入門テキスト（本科目のリポジトリの `doc` ディレクトリ内の `mltext.pdf`）が参考になるであろう．

OCaml力をチェックするための問題

本書中の実装課題に取り組む場合，以下の問題を解ける程度の OCaml 力が必要である．

Exercise 1.0.1 *OCaml*インタプリタに以下の入力を与えたところ，

```
# let rec f x = if x = 0 then x else false;;
```

Error: <u>This</u> expression has type bool but an expression was expected of type int.　という応答が返ってきた．この応答の意味するところを，エラーメッセージ中の下線部が付された *This* が何を指すかを明らかにしつつ，説明せよ．

Exercise 1.0.2

1. 各ノードに `int` 型の値を保持する二分木を表すユーザ定義型 `bt` を，ヴァリアント型を用いて定義せよ．
2. `bt` 型の値 t を受け取り，t 中に現れるすべての値の和を求める関数 `sumtree` を書け．`sumtree` の型は `bt`→`int` となる．
3. `bt` 型の値 t と `int`→`int` 型の関数 f とを受け取り，t 中に現れるすべての値に f を適用して得られる木を求める関数 `mapTree` を書け．`mapTree` の型は (`int`→`int`)→`bt`→`bt` となる．

OCaml のインストールと設定

本書の演習問題に取り組むためには，OCaml の処理系と，元になるソースコードが必要となる.[1]OCaml の処理系は，OCaml のパッケージマネージャである `opam` を用いて導入するのをおすすめする．テキスト執筆日現在，`opam` のインストール方法は `ocaml.org` の `http://www.ocaml.org/docs/install.html` や，`opam` のウェブサイト中の `https://opam.ocaml.org/doc/Install.html` に書いてあるので，各自読みながらインストールされたい.[2]インストール後に `opam init` を実行するのを忘れないこと．また，`opam` が環境変数を操作した後には，シェルを再起動するか，`opam config env` を実行して環境変数を再度シェルに読み込ませる必要がある.[3]

`opam` を用いて様々な OCaml のライブラリが利用できるのだが，本書の演習問題で使うのは `menhir` のみである．シェルで `opam install menhir` とタイプしてインストールしよう.[4]

OCaml には広く使われている統合開発環境は無いので，好みのエディタでプログラムを書いて，それをビルドして実行する，という形で開発が進むことになる．どのような言語であっても，快適に開発をするためには，キーワードをハイライトしたりインデントを自動調整したりできるようにエディタを設定することが重要である．Emacs の場合は `tuareg-mode` というモードが `opam install tuareg && opam user-setup install` で入るはずである．

また，OCaml ソースコード中の変数や式の型を簡単なショートカットで表示するための `merlin` というライブラリを入れるのがとてもおすすめである．`Merlin` のリポジトリは `https://github.com/ocaml/merlin` にあり，README にインストール方法が載っている．Emacs と Vim については公式でサポートされており，その他のいくつかのエディタについては README に記述がある．`opam install merlin && opam user-setup install` でインストールされるはずである．

---

[1]計算機科学コースの演習室の計算機環境には，すでに必要な環境設定が施されているので，この節はスキップしてもよい．

[2]Windows 環境で opam が公式にサポートされているかどうか，テキスト執筆時点ではよく分からなかった．できれば Windows 以外の環境で環境構築するのがよいであろう．

[3]ライブラリが見つからない系のエラーが出たら，とりあえず `opam config env` を実行する，とおぼえておくとよい．

[4]他の言語処理系同様，opam も各自のホームディレクトリにライブラリをインストールする．opam はデフォルトで `~/.opam/system/lib/` 以下にライブラリをインストールする．また，opam には幾つかのバージョンのコンパイラをスイッチする `opam switch` というコマンドがあるのだが，これを用いてデフォルト以外のバージョンのコンパイラをインストールした場合，このパスは `~/.opam/<version of OCaml>/lib/` になる．

第2章　概論的な話

「新しい字を発明しました。」「………どう読むのかね？」

吉田戦車「伝染るんです」

2.1　プログラミング言語を定義する

プログラム (*program*) とは，計算機に行わせるべき処理を記述した文字列である．世の中には様々な**プログラミング言語** (*programming language*) が存在する．(なお，以下では「言語」と言ったら，特に断らない限りプログラミング言語を意味する．) 例えば，

```
int main(void) {
  printf("%d", 3);
  return 0;
}
```

は標準出力に"3" という文字列を表示する C 言語のプログラムであり，

```
let x = 3 in
  x + 2
```

は x が束縛された整数値 3 から整数値 5 を計算する OCaml のプログラムである．ソフトウェアを作る際には，様々な制約[1]を考慮してどの言語を用いるかを決定することになる．

この講義で扱うのは，これらの言語を使う方法ではなく，**作る**方法である．言語を作るにはどうすればよいかを説明する前に，プログラミング言語を作るとはどういうことかを説明しよう．プログラミング言語を作るとは，言語の**シンタックス（統語論）** (*syntax*) と**セマンティクス（意味論）** (*semantics*) を定義することである．[2]シンタックスは大まかに言えば言語

[1]例えば「解決すべき問題に適した言語機構があるか」「使えるライブラリは豊富にあるか」「その言語を使える開発者が十分にいるか」「その言語はバズっているか」「おれはこの言語が好きだ」「おれはこの言語が嫌いだ」などがある．ところで，後ろ 3 つはジョークのつもりで書いたが，(ウケという意味でも，笑い話でなく実際にこうなのではないかという意味でも）あまり良いジョークになっていないかもしれない．

[2]シンタックスとセマンティクスは言語のためだけの概念ではなく，言語学や論理学など，言語を扱うより広い学問分野で生まれた概念である．プログラミング言語のように人工的に作られた言語に対し，日本語，英語，ポルトガル語などの人間が自然に使い始めた言語を**自然言語** (*natural language*) と呼ぶ．自然言語においてもシンタックスとセマンティクスが考えられるが，ほとんどの自然言語においてそれらはプログラミング言語のようにクリアに定まるものではない．また，自然言語においてはプログラミング言語では考慮されることの少ない「音韻」や「発話者等の言葉が発せられた文脈」等も言語を構成する重要な要素である．これらに興味がある場合は，自然言語処理や言語学の教員を誰か捕まえて読むべき本を教えてもらうと良いであろう．京都大学では教員に疑問をぶつけるためのサービス mondo [] が提供されているので，それらを使ってもよいかもしれない．

の語彙と文法のことであり，どのような文字列が文法的に正しいプログラムと言えるかを決定する言語の要素である.[3]例えば,

```
int main(void) { int x = 3; return 0; }
```

は文法的に正しい C 言語のプログラムであるが,

```
public class Main { static public void main(String[] argv) { int x = 3; return; } }
```

は（大体同じように振る舞う文法的に正しい Java のプログラムであるが）文法的に正しい C 言語のプログラムではない．また,

```
I hereby define a function named main.  This
function takes no arguments and returns an integer value.  The
function main first allocates a space on the stack enough to
accomodate an integer value and initializes it with an integer value
3.  Then, main returns 0 to the caller.
```

は（定義しようとしている関数の振る舞いが人間に正しく伝わるものの）やはり文法的に正しい C 言語のプログラムではない．言語のシンタックスは**文脈自由文法** (*context-free grammar*) やその変種である **BNF**(*Backus-Naur form*) 等で定義される．詳細は「プログラミング言語」の講義資料を復習されたい.

セマンティクスは大まかにはプログラムの意味のことである．例えば先程の C 言語のプログラム

```
int main(void) { int x = 3; return 0; }
```

は「変数 x に整数 3 を代入し 0 を返す」という計算を行うということが，C 言語の仕様が定めているセマンティクスから分かる.

シンタックスは文脈自由文法や BNF 等で定められると上に書いたが，それではセマンティクスはどのように定められるのだろうか．もっとも一般的な方法は自然言語による定義である．例えば C 言語の言語仕様ドラフト [] の 6.5 節には C 言語の式のセマンティクスが載っており，これを読み解くと文法に沿った C 言語のプログラムがどのような計算をすべきかが分かる．これに対し，数学的に厳密にセマンティクスを定義する方法もある．興味のある読者は五十嵐 [] や Winskel [] の教科書を読んでみるとよいだろう.

2.2　プログラミング言語の実装方式

言語のシンタックスをセマンティクスを定めると，一応プログラミング言語を作ったことになる．しかしながら，これだけでは計算機を使ってプログラムに実際に計算を行わせることはできない．計算機でプログラムを動かすには，セマンティクスに従ってプログラムを実行するためのソフトウェアが必要である．このようなソフトウェアを**言語処理系** (*implementation of a programming language*) と呼ぶ.

---

[3]このテキストでは「シンタックス」と「セマンティクス」という言葉が大量に出てくる．これらの言葉がテキストを理解する上での妨げになるようであれば,「シンタックス」を「文法」と読み替え,「セマンティクス」を「意味」と読み替えてもそんなに間違いではない.

言語処理系の例を見てみよう．コンピュータに GCC がインストールされているとしよう．その上で，先程のプログラム

```
int main(void) { int x = 3; return 0; }
```

を `main.c` という名前で保存し，シェルに

```
> gcc -o main main.c
```

と入力して実行すると，`main` というファイルができる．このファイルは実行可能なバイナリであり，シェルに

```
> ./main
```

と入力して実行することができる．(このプログラムは入出力を行わないので，プログラムを実行しても傍目には何も起こらない．) ここで `gcc` は，C 言語のシンタックスに沿ったプログラムを受け取り，そのプログラムと（C 言語のセマンティクスの上で）同等の動きをする別のプログラム（実行可能バイナリ）を生成した．このように，**入力プログラムと同じ動きをする別のプログラムを生成する**ことで，プログラミング言語を実装することができる．

別の例を見てみよう．OCaml 処理系がインストールされている計算機でシェルを立ち上げて

```
> ocaml
```

と入力してみよう．すると，インストールされている OCaml のバージョンが表示されたあとに，

```
#
```

という文字が表示されて入力待ち状態になる．(`#`のような入力待ち時に表示される文字を一般に**プロンプト** (*prompt*) と言う．) ここで

```
# let x = 3;;
```

と入力してエンターキーを押すと

```
val x : int = 3
#
```

と画面に表示され，再びプログラムの入力待ち状態になる．画面に表示された文字列は「整数型の値 3 が計算され，変数 x が計算結果に束縛された」ことを表現している．`ocaml` コマンドは (1) ユーザから文字列を受け取り，(2) その入力をプログラムとして解釈して，(3) 必要であれば自分自身の状態を変更して（ここでは変数 x を 3 という名前にして）(4) 計算結果を表示するというループを繰り返している．このように，**受け取ったプログラムを解釈してセマンティクスに従って実行し計算結果を出力する**プログラムによってプログラミング言語を実装することができる．(GCC が出力していたのは計算結果ではなく，計算を行う別のプログラム（実行可能バイナリ）であったことに注意されたい．) なお，上記の (1)–(4) のループには **read-eval-print ループ** (*read-eval-print loop*) という名前がついている．

なお，上記の 2 つの方式は，両方組み合わせて使われることがある．`ocaml` コマンドを

```
> ocaml -dinstr
```

というオプション付きで起動してみよう．すると，なにやらたくさん文字が表示された後にプロンプトが表示される．先ほどと同様に `let x = 3;;` と入力してエンターキーを押すと

```
const 3
push
acc 0
push
const "x"
push
getglobal Toploop!
getfield 1
appterm 2, 4
```

のような文字が表示される．(インストールされている OCaml のバージョンによって表示される情報は変わりうる．) この文字列は，OCaml バイトコードという，オリジナルの OCaml プログラムよりもマシン語に近い別のプログラミング言語で書かれたプログラムの文字列表現である．すなわち，`ocaml` コマンドは入力された式を，それと同じセマンティクスを持つバイトコードに内部で一旦変換し，それを解釈することで式の評価を行っている．このように，前者の実装方式（プログラムを変換する方式）と後者の実装方式（プログラムを解釈実行して結果を返す方式）とは相反するものではなく，組み合わせて使われることもある．

代表的な言語処理系の実装方法には**インタプリタ** (*interpreter*) と**コンパイラ** (*compiler*) がある．インタプリタとは，プログラムを解釈し，言語のセマンティクス従って計算を行い，その実行結果を返すソフトウェア，コンパイラとはプログラムを別のプログラムやバイナリに変換するソフトウェアである．(この変換のことを**コンパイル** (*compile*) という．) 上述した GCC はコンパイラの例で，`ocaml` コマンドはコンパイラとインタプリタを組み合わせた例である．(このように，受け取ったプログラムをコンパイルしてから実行するプログラムを**インタラクティブコンパイラ** (*interactive compiler*) と呼ぶ．

コンパイラ言語とインタプリタ言語 Web 上の情報やプログラミングに関する書籍の中には，主要な言語処理系がコンパイラとして実装されている言語をコンパイラ言語，主要な言語処理系がインタプリタとして実装されている言語をインタプリタ言語としたものが見られる．しかしながら，言語の定義と，その言語の処理系をどのように実装するかは別の問題であり，また上で見たようにコンパイラとインタプリタが一つの処理系に共存できないわけでもないので，言語がコンパイラ言語とインタプリタ言語に分類できるというわけではない．末永は個人的には「コンパイラ言語」「インタプリタ言語」という言葉には強い違和感を覚える．

2.3　中身の話

⟵
⟵

図 ??に一般的なインタプリタの流れ図を，図 ??に一般的なコンパイラの流れ図を示した．流れ図中の各要素の詳細は講義で扱うが，ここでは簡単に各要素の働きを説明する．

2.3.1　字句解析・構文解析

インタプリタに対してもコンパイラに対しても，プログラムは文字列として与えられる．処理に先立って，この文字列の文法構造を解析する必要がある．例として先程挙げたOCamlのプログラム

```
let x = 3 in
  x + 2
```

を考える．このフェーズの目的は，上の（文字列として与えられた）プログラムから以下の**抽象構文木** (*abstract syntax tree*) を得ることである．インタプリタやコンパイラは，得られた抽象構文木を用いてそれぞれの処理を行う． ⟵

抽象構文木の構築は通常**字句解析** (*lexing*) と**構文解析** (*parsing*) という二段階の処理で行われる．字句解析は，与えられた文字列を（プログラミング言語における）単語列に切り分ける処理である．例えば，上記のプログラムを字句解析器に入力すると，

$$\mathbf{LET}, \mathbf{ID}(''x''), \mathbf{EQ}, \mathbf{INT}(3), \mathbf{IN}, \ldots$$

のようなトークン列が出力される．ここで，**LET**，**ID**$(''x'')$，**EQ** **INT**(3) 等はそれぞれプログラム中の `let`，x，$=$, 3 等の単語に対応するデータ型の値である．このようなプログラム言語の「単語」を**トークン** (*token*) と呼ぶ．トークンには別のデータが付帯していてもよい．上記の例では **ID** はプログラム中で用いる（変数名や型名などの）名前（**識別子** (*identifier*)）に対応するトークンであるが，このトークンにはプログラム中で出現した実際の名前（”fact” や”n” など）がデータとして付帯している．また，**INT** はプログラム中の整数定数に対応するトークンであるが，実際の整数値を付帯データとして持っている．

構文解析は，字句解析によって切り出された[4]トークン列を抽象構文木に組み立てる処理である．トークン列から言語の文法に沿った抽象構文木を得る（あるいは，文法に沿っていなければエラーを報告する）ためのアルゴリズムが多く使われており，本講義ではそれらの一部について解説する予定である．

本講義の後半では字句解析や構文解析のアルゴリズムを扱うが[5]自分の言語処理系で字句解析や構文解析を使う際にこれらのアルゴリズムを一から実装することはほとんどない．**パーザジェネレータ** (*parser generator*) というツールは BNF で書かれた文法の定義を入力として，構文解析器を出力する.[6]このツールを使えば字句解析器や構文解析器を比較的容易に作ることができる．よく使われているパーザジェネレータには Yacc や Bison がある．また，OCaml では Menhir と呼ばれるツールがよく用いられる．本講義ではこれらのツールの使い方も扱う.[7]

[4]字句解析によって文字列としてのプログラムからトークン列を得ることをよく「切り出す」と言う．切り出し感がどこから出てくるのかはよく分からない．

[5]ちなみに，これらのアルゴリズムを理解するには,「言語・オートマトン」で学んだ文字列に対する有限状態オートマトン，正則言語，文脈自由文法，文脈自由言語の理論を理解している必要がある．理解が行き届いていない場合は，各自復習されたい．

[6]パーザジェネレータは BNF という言語で書かれた文字列を入力として構文解析を行うプログラムを出力するので，コンパイラの一種である．

[7]ツールの使い方さえ分かれば字句解析や構文解析のアルゴリズムなど学ぶ必要はないではないかという（末

2.3.2　インタプリタの中身

インタプリタにおいては，構文解析器によって生成された抽象構文木をなぞりながら，言語のセマンティクスにしたがって計算を行う．例えば，先に挙げたOCamlのプログラムを考えてみよう.(「プログラミング言語」で学んだ）OCamlのセマンティクスによれば，このプログラムは

1. 式3を評価して値3を得て，
2. 変数xを得られた値3に束縛して，
3. `in`の後の式`x`$+$2を評価する．

インタプリタにこの計算をさせるには，生成された構文木を**再帰的に**たどればよい．すなわち，式を計算する関数を`eval_exp`とすると，

1. Rootノードにぶら下がっている式3を`eval_exp`によって**再帰的に**計算し，計算結果3を得て，
2. Rootノードに識別子`x`がぶら下がっているので，変数xを3に束縛し，
3. Rootノードにぶら下がっている別の式`x`$+$2を`eval_exp`によってx**が**3**に束縛されていることを考慮しつつ再帰的に**計算する，

ようにすればよい．

インタプリタを実装する際の基本は構文木を上記のように再帰的にたどることである．ただし

- どのような順番で構文木をたどるべきか（この順番のことを**評価戦略**(*evaluation strategy*)という）
- 変数が束縛されている値をどのように保持するか
- 高階関数等の複雑なデータをどのようなデータ構造で保持するか

などを考える必要がある．本書ではこれらの詳細について実際に小さなインタプリタを作成しながら学ぶことになる．

永も学生時代に持った）問については，以下のいくつかの答えが考えられる．(1) 単位はほしいかね？(2) これらのツールは万能ではなく，人間が背景理論を理解した上でチューニングをしなければならない場合が多々ある．例えば，入力された文法が曖昧であった場合（すなわち，同一の文字列に対して2つ以上の抽象構文木がありえる場合）には，どちらが意図された構文木であるかをツールが理解することはできない．この場合には，パーザジェネレータが出力した警告を読んで，文法の曖昧性を解消する必要がある．そのためには構文解析の理論を理解する必要がある．(3) 字句解析・構文解析のアルゴリズムは，オートマトンの理論をとてもうまく用いて作られている．このアルゴリズムを学ぶことは，他のアルゴリズムを設計する場合にも有用である．

2.3.3 コンパイラの中身

コンパイラも構文木をたどりながら処理を行う点はインタプリタと共通である．ただし，インタプリタがセマンティクスに従って計算を行っていたのに対し，コンパイラは自身が計算をするのではなく,「セマンティクスに従って計算を行う変換先言語のプログラム」を生成する．

再び先程のプログラム `main.c` を gcc を用いてコンパイルしてみよう．ただし，今度は以下のように gcc を起動してみよう．

```
> gcc -S -o main.s main.c
```

gcc は上のように`-S` オプション付きで起動されると，入力プログラム（ここでは `main.c`）を実行可能バイナリではなく，**アセンブリ** (*assembly*)，すなわち実行可能バイナリを人間が読みやすい文字列表現にしたファイルにコンパイルする．生成されたアセンブリは`-o` の後に指定されたファイル（ここでは `main.s`）に記録される．著者の環境でのこのファイルの内容は以下の通りである．

このファイルの内容を理解する必要はないのだが，重要なのはアセンブリも**アセンブリ言語** (*assembly language*) という言語で書かれたプログラムの一種であるということである．アセンブリ言語は計算機の命令を一つ一つ指定することで行うべき計算を記述するための言語で，様々な計算機アーキテクチャに固有のアセンブリ言語が定義されている．

アセンブリは**アセンブラ** (*assembler*) というコンパイラを用いて実行可能バイナリにコンパイルすることができる．gcc は実はアセンブラとしての機能も持っており，拡張子.s を持つファイルをアセンブリと仮定する．したがって，生成された `main.s` を

```
> gcc -o main main.s
```

のように gcc に与えれば，実行可能バイナリ main が得られ

```
> ./main
```

と実行することができる．

少し長くなったが，ここでの要点はインタプリタはプログラムを受け取って，言語のセマンティクスに従って計算を行い，計算結果を返すのに対し，コンパイラはプログラムを受け取って，言語のセマンティクスに従って計算を行う別のプログラムを返すという点である．変換元の言語（上の例では C 言語）を**ソース言語** (*source language*) と呼び，変換先の言語（上の例ではアセンブリ言語）を**ターゲット言語** (*target language*) と呼ぶ.[8]

したがって，コンパイラを実装するためには，構文解析器が作った構文木をたどりながら，入力されたプログラムに対応するターゲット言語のプログラムを生成すれば良い．しかし，ほとんどのコンパイラはいきなりターゲット言語のプログラムを生成するのではなく，間にいくつかの**中間言語** (*intermediate language*) をはさみながら変換を行う．また，各中間言語のレベルにおいて，プログラム解析を行い，生成されるプログラムの実行効率を上げるため

---

[8]本書では，実行可能バイナリもフォーマットも広い意味ではプログラミング言語であると考えている．したがって，C プログラムを入力として受け取り実行可能バイナリを生成する gcc は，ソース言語が C 言語，ターゲット言語が「実行可能バイナリを記述するためのプログラミング言語」のコンパイラであると捉える．

の変換（**最適化** (*optimization*) と呼ばれる）を行ったり，プログラム中に存在するバグの発見を試みたりすることが多い．これらについては講義中で扱う．

2.4 本書のこれ以降の構成

以上，本書で扱うトピックを概論的に説明した．これ以降では，それぞれのトピックを掘り下げて説明する．本書のこれ以降の構成は以下の通りである．[9]

- 3章では，OCaml のサブセットを解釈実行するインタプリタの実装を説明する．この章を一通りやれば，言語を設計し実装するための基礎が身につくはずである．
- 4章では静的プログラム解析という，プログラムを実行することなくプログラムの実行に関する情報を得るための手法を解説する．こういうと結構おどろおどろしいが，実際には OCaml の型推論のメカニズムを説明する．型推論は，プログラムの実行前に実行時型エラーが無いかどうかを解析するという意味で，静的なプログラム解析である．
- 5章では，方向性が少し変わって，コンパイラの実装について簡単に解説する．ここでは，3章で定義した言語（のさらにサブセット）をコンパイラとして実装してみる．同じ言語をインタプリタとしてもコンパイラとしても実装してみることで，両者の違いと利害得失が分かるようになるであろう．
- 6章では，ここまでブラックボックスとして扱ってきた，字句解析器と構文解析器と，それらを生成するツールの中身について解説する．

[9] ただし，これは4月に締め切りに追われてヒーヒー言いながら書いている予定であり，変更される可能性は結構ある．

第3章　MLインタプリタの設計と実装

そこでみなさん，プログラムを何か一つ書いてください．私がそれを構文木にしますから，そこでなにか一言．はい楽さん早かった．

この章では，言語を定義し，そのインタプリタを実装する方法を解説する．2 章で述べたように，インタプリタは，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，そのセマンティクスに従って実行結果を計算するソフトウェアである．

セマンティクスの定義 （以下の説明は分からなければ飛ばして良い．）2 章では言語にはシンタックスとセマンティクスが定められており，インタプリタはシンタックスに従って字句解析と構文解析を行い，セマンティクスに従って計算を行うと述べた．この関係を逆に見て，インタプリタがプログラミング言語のシンタックスとセマンティクスを定義していると見ることがある.[1]すなわち，インタプリタの構文解析器の挙動をもって言語のシンタックスとし，インタプリタの計算の過程をもって言語のセマンティクスとすることもできる．この視点の下では,「言語のシンタックスとセマンティクスを与える」とは，すなわちその言語のインタプリタを一つ与えることである．実際に，数学的に厳密な形でセマンティクスを与える方法の一つである**操作的意味論** (*operational semantics*) は，インタプリタの挙動を数学の言葉（写像や関係など）で表現する．これに対置されるのが，インタプリタの挙動を経由せずに，構文木からその木が表現する計算への写像を与えることでセマンティクスを与える手法である．これを**表示的意味論** (*denotational semantics*) と呼ぶ．

コンパイラもまたこの視点から見ることができる．すなわち，コンパイラは，入力プログラムを受け取ると，そのプログラムが表す計算を実行するターゲット言語のプログラムを出力するのであるが，出力されたプログラムの挙動を通じて，ソース言語のセマンティクスが定まると見るのである．

なにを訳の分からないことをフニャフニャと言っておるのかおまえはフニャコフニャ夫かと言いたくなるかもしれないが,「プログラミング言語のセマンティクスをどのように定義するか」は**プログラミング言語理論** (*programming language theory*) と呼ばれる分野の大きなトピックの一つになっている．実行が止まらないかもしれない言語，非決定的実行を含む言語，確率的挙動を含む言語，並行性を含む言語，連続的挙動を含む言語，量子プログラミング言語など，さまざまな言語に対して数学的に厳密な形でセマンティクスを与えることは,「計算」の本質という理論的興味からも，プログラミング言語を明確かつ簡潔に定義してプログラマが安心してプログラムを作れるようにするという意味でも重要である.[2]興味があれば，五十嵐 []，Winskel [] を読むと良い．

---

[1]いわゆる「その挙動は仕様です」である．

[2]ただし，末永は主に言語に対する個人的なフェティシズムを満足させるためにこういう研究を行っている

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，

定義する言語 = OCaml
定義される言語 = ML（OCaml のサブセット）

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．

本書では，とてもシンプルな OCaml のサブセット ML^1 のインタプリタを実装し，これに以下のように徐々に言語機能を拡充していく．

ML^1 整数，真偽値，条件分岐，加算乗算とあらかじめ定義されている変数の参照のみが可能な OCaml のサブセット．

ML^2 ML^1 を変数定義機能で拡張した言語．

ML^3 ML^2 を（高階）関数で拡張した言語．

ML^4 ML^3 を再帰関数で拡張した言語．

ML^5 ML^4 をリストで拡張した言語．

3.1 プログラムファイルの構成・コンパイル方法

これから作成するインタプリタのソースコードは以下のファイルから構成される．[3]

`syntax.ml` 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．

フシがある．プログラムによって面白いソフトウェアを作ることよりも，プログラムを記述するための言語自体に興味を持つという性格は，ある種の偏愛性を帯びているものかもしれない．

[3] プログラムはこのリポジトリの `src` ディレクトリに格納されている．

`parser.mly` OCaml のパーザジェネレータ Menhir に入力する文法定義である．Menhir は `.mly` という拡張子のファイルに記述された BNF 風の文法定義から，構文解析プログラムを生成する．定義の書き方は 6 章で説明する．

`lexer.mll` OCaml の字句解析器生成ツールである ocamllex の定義ファイルである．ocamllex は `.mll` という拡張子のファイルに定義されたパターン定義から，字句解析プログラムを生成する．定義の書き方は 6 章で説明する．

`environment.mli`, `environment.ml` インタプリタ・型推論で用いる，**環境** (*environment*) 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．

`Makefile` インタプリタをビルドするために使う Makefile である．

また，以下のファイルは 4 章において，**型推論** (*type inference*) アルゴリズムを実装するために用いる．

`mySet.ml`, `mySet.mli` 集合の抽象データ型の定義とインターフェースである．型推論の実装で用いる．

`typing.ml` 型推論アルゴリズムの実装である．最初は何も書いていない．

インタプリタのビルド方法はインタプリタのソースコードが格納されているディレクトリの README.md ファイルに書いてある．このファイルはテキストファイルなので，テキストエディタで開いて読んでほしい．シェルを立ち上げて，ソースコードが格納されているディレクトリで make と入力すればビルドできるようになっている．ML というバイナリができるので，以下のように実行するとインタプリタを立ち上げることができる．

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

インタプリタの起動時にはデフォルトでいくつか変数が定義されている．起動時に大域変数 x の値は 10 になっている．ただし，デフォルトのインタプリタの機能は限られており，変数や（再帰）関数を定義することはできない．例えば，インタプリタに以下の変数定義式を入力してみよう．

```
# let a = 2 in a + 3;;
Fatal error: exception Parser.Basics.Error
```

インタプリタがエラーを吐いて落ちてしまった．出力されたメッセージから，インタプリタが `Parser.Basics.Error` という例外を投げて落ちたことが分かる．これは構文解析器の中で定義されている例外で，文法エラーが起こったために処理を続行できなかったことを表している．これから，様々な式を処理できるようにインタプリタを拡張する．

3.1.1　デバッグの仕方

⟵

3.2　ML¹ 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（新しい変数の定義すらできない!）を持つ言語 ML¹ から始める．

3.2.1　ML¹ のシンタックス

⟵　ML¹ のシンタックスは以下の BNF で与えられる．

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

ML¹ のプログラムは，;; 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 ML¹ 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．

```
(* 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 3.1: ML[1] インタプリタ: syntax.ml

3.2.2 各モジュールの機能

実装するインタプリタは 6 つのモジュールから構成される.[4]それぞれのモジュールについて簡単に説明する.

3.2.3 Syntax モジュール：抽象構文のためのデータ型の定義

Syntax モジュールはファイル syntax.ml に定義されており，抽象構文木を表すデータ型を定義している．具体的には，このモジュールでは上の BNF に対応する抽象構文木を表す以下の型定義が含まれている．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，変数の型や変数が現れたファイル名と行数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.

[4] OCaml を含め多くのプログラミング言語には，**モジュールシステム** (*module system*) と呼ばれる，プログラムを部分的な機能（**モジュール** (*module*)）ごとに分割するための機構が備わっている．この機構は，プログラムが大規模化している現代的なプログラミングにおいて不可欠な機構であるが，その解説は本書の範囲を超える.「OCaml 入門」の該当する章を参照されたい．さしあたって理解しておくべきことは，(1) OCaml プログラムを幾つかのファイルに分割して開発すると，ファイル名に対応したモジュールが生成されること（例えば，foo.ml というファイルからは Foo というモジュールが生成される）(2) モジュール内で定義されている変数や関数をそのモジュールの外から参照するにはモジュール名を前に付けなければならないこと（例えばモジュール Foo の中で定義された x という変数を Foo 以外のモジュールから参照するには Foo.x と書く）の二点である.

3.2.4 Parser モジュール，Lexer モジュール：字句解析と構文解析

Parser と Lexer はそれぞれ構文解析と字句解析を行うモジュールである．Parser モジュールは Menhir というツールを用いて parser.mly というファイルから，Lexer モジュールは ocamllex というツールを用いて lexer.mll というファイルからそれぞれ自動生成される．

Menhir は **LR(1) 構文解析** (*LR(1) parsing*) という手法を用いて，BNFっぽく書かれた文法定義（ここでは parser.mly）から，構文解析を行う OCaml のプログラム（ここでは parser.ml と parser.mli）を自動生成する．また，ocamllex は**正則表現** (*regular expression*) を使って書かれたトークンの定義（ここでは lexer.mll）から，字句解析を行う OCaml のプログラム（ここでは lexer.ml）を自動生成する．生成されたプログラムがどのように字句解析や構文解析を行うかはこの講義の後半で触れる．そのような仕組みの部分を抜きにして，ここでは .mly ファイルや .mll ファイルの書き方を説明する．

文法定義ファイルの書き方

拡張子 .mly 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 OCaml のプログラムを書く部分で，Menhir 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．コメントは OCaml と同様 (* ... *) である．[5]

それでは parser.mly を見てみよう (図 3.2).[6] 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 Syntax 宣言はモジュール Syntax 内で定義されているコンストラクタや型の名前を，Syntax. というプレフィクス無しで使うという OCaml の構文である．(これがないと，例えばコンストラクタ Var を参照するときに Syntax.Var と書かなくてはならない．[7])

[5] ヘッダ部分とトレイラ部分以外では /* ... */ と //... が使えるらしい．

[6] 以降の話は結構ややこしいかもしれないので，全部理解しようとせずに，parser.mly と lexer.mll を適当にいじって遊ぶ，くらいの気楽なスタンスのほうがよいかもしれない．

[7] OCaml 以外にもこの手の機構が用意されていることが多い．例えば Java ではパッケージの import，Python では import 文がこれに相当する．なお，open はモジュール内の名前に容易にアクセスすることを可能にするが，外のモジュールで定義されている名前との衝突も起きやすくするという諸刃の剣である．この辺の話は時間があれば講義で少し触れる．

```
%
open Syntax
%

%token LPAREN RPAREN SEMISEMI
%token PLUS MULT LT
%token IF THEN ELSE TRUE FALSE

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

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

toplevel :
    e=Expr SEMISEMI { Exp e }

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

LTExpr :
    l=PExpr LT r=PExpr { BinOp (Lt, l, r) }
  | e=PExpr { e }

PExpr :
    l=PExpr PLUS r=MExpr { BinOp (Plus, l, r) }
  | e=MExpr { e }

MExpr :
    l=MExpr MULT r=AExpr { BinOp (Mult, l, r) }
  | e=AExpr { e }

AExpr :
    i=INTV { ILit i }
  | TRUE   { BLit true }
  | FALSE  { BLit false }
  | i=ID   { Var i }
  | LPAREN e=Expr RPAREN { e }

IfExpr :
    IF c=Expr THEN t=Expr ELSE e=Expr { IfExp (c, t, e) }
```

図 3.2: ML[1] インタプリタ: parser.mly

- %token ⟨ トークン名 ⟩ . . . は，**属性** (*attribute*) を持たないトークンの宣言である．属性とは，トークンに関連付けられた（以下で説明する）還元時アクションの中で参照することができる値のことである．属性を持つトークンを見ればなるほどと納得が行くかもしれない．parser.mly 中では括弧 “(”，“)” 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 3.1 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．) この宣言で宣言されたトークン名は Menhir 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[8]）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．Menhir 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 ⟩ :
  (⟨ 変数名 $_{11}$⟩=)⟨ 記号名 $_{11}$⟩ . . . (⟨ 変数名 $_{1n_1}$⟩=)⟨ 記号名 $_{1n_1}$⟩
  { ⟨ 還元時アクション $_1$⟩ }
  | (⟨ 変数名 $_{21}$⟩=)⟨ 記号名 $_{21}$⟩ . . . (⟨ 変数名 $_{2n_2}$⟩=)⟨ 記号名 $_{2n_2}$⟩
  { ⟨ 還元時アクション $_1$⟩ }
  . . .

  のように記述する．⟨ 記号名 ⟩ の場所にはそれぞれ非終端記号か終端記号を書くことができる．「⟨ 変数名 ⟩=」の部分は省略してもよい．⟨ 還元時アクション ⟩ の場所には OCaml の式を記述する．

  構文解析器は，開始記号から始めて，与えられたトークン列を生成するために適用すべき規則を適切に発見し，それぞれの規則の還元時アクションを評価して，評価結果を規則の左辺の非終端記号の属性とすることで，開始記号の属性を計算する．と言われてもよく分からないと思うので，図 3.2 の文法定義を例にとって説明する．この文

[8] 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．

法定義から生成される構文解析器に TRUE SEMISEMI というトークン列が与えられたとしよう.[9]このトークン列は開始記号 toplevel から始めて以下のように規則を適用すると得られることが分かる.[10]

```
toplevel
--（規則 toplevel: Expr SEMISEMI を用いて）-->
Expr SEMISEMI
--（規則 Expr: LTExpr を用いて）-->
LTExpr SEMISEMI
--（規則 LTExpr: PExpr を用いて）-->
PExpr SEMISEMI
--（規則 PExpr: MExpr を用いて）-->
MExpr SEMISEMI
--（規則 MExpr: AExpr を用いて）-->
AExpr SEMISEMI
--（規則 AExpr: TRUE を用いて）-->
TRUE SEMISEMI
```

各ステップで規則が適用された非終端記号に下線を付した．各ステップで用いられた規則を確認してほしい.

構文解析器は，この導出列を遡りながら，還元時アクションを評価し，各規則の左辺にある非終端記号の属性を計算する．例えば，

```
AExpr SEMISEMI
--（規則 AExpr: TRUE を用いて）-->
TRUE SEMISEMI
```

の規則が適用されている場所では，左辺の非終端記号 AExpr の属性が還元時アクション BLit true の評価結果（すなわち，BLit true という値）となる．ここで計算された属性は，その一つ手前の導出

```
MExpr SEMISEMI
--（規則 MExpr: AExpr を用いて）-->
AExpr SEMISEMI
```

で MExpr の属性を計算するのに使われる．ここで図 3.2 の対応する規則の右辺は e=AExpr となっているが，これは先程計算した AExpr の属性を e という名前で還元時アクションの中で参照できることを表している．ここでは還元時アクションは e なので，MExpr の属性は e，すなわち AExpr の属性である BLit true となる．これを繰り返すと，開始記号 toplevel の属性が Exp (BLit true) と計算され，これがトークン列 TRUE SEMISEMI に対する構文解析器の出力となる.

[9]このトークン列は true;; という文字列を lexer.mll から生成される字句解析器に与えることで生成される.

[10]ちなみに，なぜこれが「分かる」のかが構文解析アルゴリズムの大きなテーマである．構文解析アルゴリズムについては講義中に扱うので，それまでは何らかの方法でこれが分かるのだと流してほしい.

図 3.2 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．

トークン定義ファイルの書き方

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，OCaml 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 (OCaml 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．

```
{
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 3.3: ML[1] インタプリタ: `lexer.mll`

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 OCaml マニュアルを参照すること．

3.2.5 Environment モジュールと Eval モジュール：環境と解釈部

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** (*value*) 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．例えば，C 言語であれば整数値，浮動小数値，ポインタなどが値として扱えるし，OCaml であれば整数値，浮動小数値，レコード，ヴァリアント，クロージャ，オブジェクトなどが値として扱える．

この時**式の値** (*expressed value*) の集合と**変数が指示する値** (*denoted value*) を区別することがある．前者は式を評価した結果得られる値であり，後者は変数が指しうる値である．この 2 つの区別は，普段あまり意識することはないかもしれないし，実際に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（式の値の集合 = 変数が指示する値の集合）．しかし，この 2 つが異なる言語も珍しくない.[11]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．そのため，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

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

と与えられる．$\oplus$ は直和を示している．

このことを表現した OCaml の型宣言を以下に示す．

```
(* Expressed values *)
type exval =
```

[11] この二種類の値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．

```
    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，変数と denoted value の束縛[12]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．

環境はここでは変数と denoted value の束縛を表現できれば充分なのだが，あとで用いる型推論においても，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，環境の型を多相型 `'a t` とする．ここで `'a` は変数に関連付けられる情報（ここでは denoted value）の型である．こうすることで，同じデータ構造を変数の denoted value への束縛としても，変数の別の情報への束縛としても使用することができるようになる．

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` は，何の変数も束縛されていない，空の環境である．次の `extend` 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 id dnval env` で，環境 `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 3.4 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．

「実装の隠蔽」について `environment.mli` と `environment.ml` の関係を理解しておくのはとても重要なので，ここで少し説明しておこう．どちらも `Environment` モジュールを定義するために用いられるファイルなのだが，`environment.ml` は `Environment`

[12] 一般に変数 x が何らかの情報 v に結び付けられていることを x が v に**束縛されている** (*x is bound to v*) と言う．

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

モジュールの**実装** (*implementation*) を定義し,[13]environment.mli はこのモジュールの**インターフェイス** (*interface*) を宣言する.[14]

これを頭に入れて，environment.ml と environment.mli を見返してみよう．environment.ml は型 'a t を連想リスト（Syntax.id * 'a） list として定義し，'a t 型の値を操作する関数を定義している．これに対して，environment.mli は (1) なんらかの多相型 'a t が**存在する**ことのみを宣言しており，この型の実体が何であるかには言及しておらず，(2) 各関数の型を 'a t を用いて宣言している.（.mli ファイル中の各関数の型宣言は 'a t の実体が（Syntax.id * 'a） list であることには言及していないことに注意.）

Environment モジュール中の定義を使用するモジュール（例えばあとで説明する Eval モジュールなど）は，environment.mli **ファイルに書かれている定義のみを，書かれている型としてのみ**使うことができる．例えば Environment モジュールの empty という変数を Environment モジュールの外から使う際には Environment.empty という名前で参照することになる．Environment.empty は 'a t 型なので**リストとして使うことはできない**．すなわち，environment.ml 内で 'a t がリストとして実装されていて empty が [] と実装されているにも関わらず，1 :: Environment.empty という式は型エラーになる．

なぜこのように実装とインターフェイスを分離する言語機構が提供されているのだろうか．一般によく言われる説明は**プログラムを変更に強くするため**である．例えば，開発のある時点で Environment モジュールの効率を上げるために，'a t 型をリストではなく二分木で実装し直したくなったとしよう．今の実装であれば，'a t 型が実際はどの

[13]すなわち，environment.ml は，Environment モジュールがどう動作するかを決定している．

[14]すなわち，environment.mli は，このモジュールがどのように使われてよいかを決定している．一般にインターフェイスとは，2 つ以上のシステムが相互に作用する場所のことを言う．Environment モジュールの内部動作と外部仕様との相互作用を environment.mli が決めているわけである．

型なのかがモジュールの外からは隠蔽されているので，environment.ml を修正するだけでこの変更を実装することができる．このような隠蔽のメカニズムがなかったとしたら，Environment モジュールを使用する関数は，'a t 型がリストであることに依存したプログラムになっているはずであるから，二分木への変更を行うためには全プログラム中の Environment モジュールを利用しているすべての箇所の修正が必要になる．この例から分かるように，実装とインターフェイスを分離して，モジュール外には必要最低限の情報のみを公開することで，変更に強いプログラムを作ることができる．

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 という関数として図 3.5 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 Environment.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 OCaml 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 ML¹ 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．

3.2.6 main.ml

メインプログラム main.ml を図 3.6 に示す．関数 read_eval_print で，

1. 入力文字列の読み込み・構文解析
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

```
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 3.5: ML^1 インタプリタ: 評価部の実装 (`eval.ml`) の抜粋

```
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 3.6: ML[1] インタプリタ: `main.ml`

`%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．これらの関数はそれぞれ ocamllex と Menhir によって自動生成された関数である．`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` 型の値を取る．標準ライブラリの `Lexing` モジュールの説明を読むと分かるが，`Lexing.lexbuf` の作り方にはいくつか方法がある．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．`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．

> **標準ライブラリ** 本書を書いている時点では，OCaml の標準ライブラリは `http://ocaml.org/` で “Documentation” → “OCaml Manual” → “The standard library” の順にリンクをたどると出て来る．このページには標準ライブラリで提供されている関数がモジュールごとに説明されている．
>
> なお，OCaml の標準ライブラリは必要最低限の関数のみが提供されているため，OCaml でソフトウェアを作る際にはその他のライブラリの力を借りることが多い．様々なライブラリをパッケージマネージャの opam を用いてインストールすることができる．

Exercise 3.2.1 *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 3.2.2 [⋆⋆] 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 3.2.3 [⋆] 論理値演算のための二項演算子 &&, || を追加せよ．

Exercise 3.2.4 [⋆⋆] `lexer.mll` を改造し，`(*` と `*)` で囲まれたコメントを読み飛ばすようにせよ．なお，OCaml のコメントは入れ子にできることに注意せよ．ocamllex のドキュメントを読む必要があるかもしれない.（ヒント：`comment` という再帰的なルールを `lexer.mll` に新しく定義するとよい.）

3.3 ML^2 — 定義の導入

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

変数宣言と有効範囲

OCaml の `let` 式は変数の定義と，その定義の下で評価される式が対になっている．例えば，以下の OCaml プログラム let x = 1 in let y = 2 + 2 in (x + y) * v は，変数 x を式 1 の評価結果（つまり整数値 1）に，変数 y を 式 2+2 の評価結果（つまり整数値 4）に束縛した上で，式 (x + y) * v を評価する，という意味である.（変数 v は初めに定義されている環境で 5 に束縛されていたことを思い出されたい.）

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

一般に，ML^2 の let 式は，

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

束縛変数 「束縛変数」という概念が末永は学生のころなかなか理解できなかった記憶がある．他にもそういう人がいるかもしれないので，一応ここで説明を加えておく．束縛変数とは直観的には「名前替えをしても意味[15]が変わらない変数」のことを言う．たとえば，let x = 3 in x + 2 という式は let y = 3 in y + 2 という式と（プログラムとしては）同じ意味を持っている．両者とも「何らかの変数を整数 2 であると定義し，その変数に 2 を加えた値を評価結果とする」という意味になっているからである.（ここで「何らかの変数」を前者の式は x としており，後者の式は y と取っている.）このように名前の付け替えをしても式の意味として変化がないときに，その名前替えをされてよい変数を束縛変数というのである．

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

また，多くのプログラミング言語と同様に，ML^2 では，ある変数の有効範囲の中に，同じ名前の変数が宣言された場合，内側の宣言の有効範囲では，外側の宣言を参照できない．このような場合，内側の有効範囲では，外側の宣言の**シャドウイング** (*shadowing*) が発生しているという．例えば，(* 一つ目の x の定義 *) let x = 2 in let y = 3 in (* 二つ目の x の定義 *) let x = x + y in x * y という ML^2 の式において，一つ目の x の定義の有効範囲は，内側の let 式全体（すなわち let y = 3 in let x = x + y in x * y）であるが，二つ目の x の定義によって一つ目の定義がシャドウイングされるので，式 x * y 中では一つ目の x の定義を参照することはできない．また二つ目の x の定義の右辺に現れる x + y の x は一つ目の x の定義を参照しているので，この式の値は 15 である．実は，最初の例でも x の宣言は，大域環境で束縛されている x のシャドウイングが発生しているといえる．

3.3.1 let 宣言・式の導入

ML^2 の構文は，以下のように与えられる．

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

Expressed value, denoted value ともに以前と同じ，つまり，let による束縛の対象は，式の値である．この拡張に伴うプログラムの変更点を図 3.7 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，式をパターンマッチで取りだし，各式を評価する．その値を使って，現在の環境を拡張し，本体式を評価している．トップレベル定義の評価（`eval_decl`）では，拡張された

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

syntax.ml:

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

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

parser.mly:

```
%token LET IN EQ

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

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

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

lexer.mll:

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

...

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

eval.ml:

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

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

図 3.7: 局所定義

Exercise 3.3.1 ML^2 インタプリタを作成し，テストせよ．

Exercise 3.3.2 [$\star\star$] OCaml では，`let` 宣言の列を一度に入力することができる．この機能を実装せよ．以下は動作例である．

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

Exercise 3.3.3 [$\star\star$] バッチインタプリタを作成せよ．具体的には `miniml` コマンドの引数として ファイル名をとり，そのファイルに書かれたプログラムを評価し，結果をディスプレイに出力するように変更せよ．また，コメントを無視するよう実装せよ．(オプション: `;;` で区切られたプログラムの列が読み込めるようにせよ．)

Exercise 3.3.4 [$\star\star$] and を使って変数を同時にふたつ以上宣言できるように let 式・宣言を拡張せよ．例えば以下のプログラム

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

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

3.4 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．

3.4.1 関数式と適用式

まずは，ML^3 の式の文法を示す．

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

構文に関するインタプリタ・プログラムは，図 3.8 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．

syntax.ml:

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

parser.mly:

```
    %token RARROW FUN

    Expr :
       ...
       | e=FunExpr { e }

    MExpr :
        e1=MExpr MULT e2=AppExpr { BinOp (Mult, e1, e2) }
      | e=AppExpr { e }

    AppExpr :
        e1=AppExpr e2=AExpr { AppExp (e1, e2) }
      | e=AExpr { e }
```

lexer.mll:

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

図 3.8: 関数と適用 (1)

3.4.2 関数閉包と適用式の評価

さて，OCaml 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．

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

さて，関数値をどのようなデータで表現すればよいか，すなわち `fun x -> e` という関数式を評価した結果をどう表現すればよいかを考えよう．この式を評価して得られる関数値は,，何らかの値に適用されると，仮引数 `x` を受け取った値に束縛し，関数本体の式 `e` を評価する．したがって，関数値は少なくともパラメータの名前と，本体の式の情報を含んでいなければならない．したがって，以下のように `exval` を拡張して関数値のためのコンストラクタ `ProcV` を定義することが考えられる．

```
(* 注：これはうまくいかない *)
type exval =
   ...
  | ProcV of id * exp
and dnval = exval
```

しかし，実際はこれだけではうまくいかない．以下の ML^3 のプログラム例を見てみよう．

```
let f =
  let x = 2 in (* (A) *)
  let addx = fun y → x + y in
  addx
in
f 4
```

この例で定義している関数 addx は受け取った値に x の値を加えて返す関数である．前述のように関数値を表現するとこの addx は `ProcV("y", BinOp(Plus, Var "x", Var"y"))` に束縛されるはずである．このプログラムは f を addx の評価結果に束縛するので，f を

```
ProcV("y", BinOp(Plus, Var "x", Var "y"))
```

に束縛した環境で関数適用式 f 4 を評価しようとする．したがって，上述した関数適用の直観的なセマンティクスによれば，変数 y を 4 に束縛して `BinOp(Plus, Var "x",Var "y")` を評価することになるのだが，この時点では環境中に x がないために（(* (A) *) の[16]x の束縛はこの時点ではスコープを外れていることに注意）このままだと正しくプログラムを評価することができない．

[16](* (A) *)ってクマみたいに見えますね．

問題は，addx が束縛される関数式 fun y → x + y に自由変数 x が含まれていることである．このx は，OCaml と同様に addx が定義された時点の x の値（すなわち，(A) の行で導入される束縛が有効）なのだが，関数値に仮引数と関数本体の式のみを含める現在の定義では，このような関数式の自由変数が扱えない．このような自由変数を扱うためには，関数値に仮引数，関数本体の式に加えて，関数値が作られたときに自由変数が何に束縛されているか，すなわち現在の例では x が 2 に束縛されているという情報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
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 3.9 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 3.4.1 *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 3.4.2 [$\star\star$] OCaml 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 3.4.3 [$\star$] OCaml の

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

といった簡略記法をサポートせよ．

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
  ...
  (* 現在の環境 env をクロージャ内に保存 *)
  | 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') ->
              (* クロージャ内の環境を取り出して仮引数に対する束縛で拡張 *)
              let newenv = Environment.extend id arg env' in
                eval_exp newenv body
        | _ -> err ("Non-function value is applied"))
```

図 3.9: 関数と適用 (3)

Exercise 3.4.4 [$\star$] 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 3.4.5 [$\star$] 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，関数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．インタプリタを改造し，`fun` の代わりに `dfun` を使った関数は動的束縛を行うようにせよ．例えば，

```
let a = 3 in
let p = dfun x → x + a in
let a = 5 in
  a * p 2
```

というプログラムでは，関数 p 本体中の a は 3 ではなく 5 に束縛され，結果は，35 になる．(`fun` を使った場合は 25 になる．)

Exercise 3.4.6 [$\star$] 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 3.4.4 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．以下のプログラムで， 二箇所の `fun` を `dfun` に置き換えて (4 通りのプログラムを) 実行し，その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
```

3.5 ML^4 — 再帰的関数定義の導入

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

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

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

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

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

おなじみの階乗関数である．この式を評価する際には，まず関数 fact を関数閉包に束縛する．この関数閉包は，n を受け取って if n = 0 then 1 else n * (fact (n - 1)) を返す関数である．この関数閉包内には，3.4.1 節で説明したとおり，関数閉包を作る時点での環境が保存される．いまこの環境はデフォルトの大域環境 `initial_env` と同じである．この関数閉包を fact 5 で使用している．関数適用を行う際には，関数閉包内に保存されている環境を取り出し，その環境を仮引数に対する束縛で拡張した上で関数本体の評価を行う．したがって，この例では，`initial_env` を n=5 で拡張した環境で if n = 0 then 1 else n * (fact (n - 1)) の部分を評価することになる．数ステップ後，インタプリタは fact (n - 1) をこの環境で評価することになるのだが，環境内には `fact` に対する束縛が含まれていないので，エラーとなる．

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

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

図 3.10 が，主なプログラムの変更点である．再帰関数を定義する際に，一旦ダミーの環境を作成し，関数閉包を作成した後に，その環境を更新する必要があるが，これを OCaml の参照を用いて実現している．`eval.ml` の `exval` 型の定義において，`ProcV` が保持するデータが環境 `dnval Environment.t` ではなく，環境への参照 `dnval Environment.t ref` になっていることに注意されたい．(したがって，ここに明示されていない関数適用のケースにおいては，格納されている環境を使用するために，参照から環境を取り出す操作が必要になる．) `eval_exp` の `LetRecExp` を処理する部分は，まずダミーの型環境への参照 `dummyenv` を作った上で，この `dummyenv` を含む関数閉包を作成し，現在の環境 `env` を `id` からこの関数閉包への写像で拡張した環境 `newenv` を作り，を上で述べたバックパッチをまさに実現している．

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

Exercise 3.5.2 [$\star\star$] 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．

syntax.ml:

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

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

eval.ml:

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

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

図 3.10: 再帰的関数定義

3.6 ML^5 and beyond — やりこみのための演習問題

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

3.6.1 リストとパターンマッチ

Exercise 3.6.1 [$\star\star$] 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，

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

程度の制限されたものでよい．

Exercise 3.6.2 [$\star$] リスト表記

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

をサポートせよ．

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

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

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

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

などが正しいプログラムとして認められるようにせよ．

3.6.2 自由課題

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

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

第4章　型システムを用いた形式検証

ここに入れるべき洒落たフレーズを思いつきませんでした．

末永幸平

4.1　能書き

3 節で実装したインタプリタは，与えられたプログラムを，セマンティクスにしたがって評価することにより実行結果を得ていた．例えば，実装したインタプリタを使って式 3+5 を評価すると 8 が評価結果として返ってくる．式 let x = 2 in x + 3 を評価すると 5 が評価結果として返ってくる．

では，プログラムの実行結果についての情報を得る方法は評価だけなのだろうか．**プログラムを実行することなく**プログラムの実行結果についてなんらかの情報を予測することは可能だろうか．この章のテーマは，プログラムを実行せずに解析して，その実行についての情報を得る方法である**静的解析** (*static analysis*) である．[1]

プログラムを実行すれば実行結果が得られるのに，なぜわざわざ静的解析をやろうとするのだろうか．静的解析の用途としては，例えば以下のものがある．

- 静的解析によって，プログラムの実行に一切影響を与えないプログラム中の部分を発見したり，常に一定の値を取る変数を発見するなどして，プログラムの実行効率を上げることができる．これらは言語処理系，特にコンパイラの文脈では**最適化** (*optimization*) として知られる処理である．

- プログラム中には「絶対に成り立っているはずというプログラマの意図」を**アサーション** (*assertion*) として記述することがある．例えば，以下の C 言語の関数 `sum` は 2 つの整数引数 `x` と `y` をとり，`x+y` を返す関数であるというプログラマの意図が `assert(ret == x + y);` という文で表現されている．[2]

[1] ここでいう「静的」とは「プログラムを実行する前に」という意味である．反対にプログラムを実行させて行う解析を**動的解析** (*dynamic analysis*) と呼ぶ．例えば，プログラムを実行して実行時間の大部分を占める（すなわち効率化をすることで効果が上がりやすい）関数を探す方法（**プロファイリング** (*profiling*)）や，プログラムを様々な入力で実行してバグを見つける方法（**ソフトウェアテスト** (*software test*)）はそのような動的解析の一種である．

[2] `assert(e)` は実行時には `e` を評価し，その結果が偽（C 言語では 0）であればエラーを報告してプログラムを終了する関数として実装されていることが多い．

```
/* Returns x+y */
unsigned int sum(unsigned int x, unsigned int y)
  unsigned int i = 0;
  unsigned int ret = x;
  while (i < y)
    ++i; ++ret;

  assert(ret == x + y);
  return ret;
```

アサーションはプログラマが「絶対に成り立つ」と表明した条件なので，これが実際に成り立っていることを保証するのは，プログラムの信頼性を向上させる上で重要である．これを保証する手段として静的解析が使われることがある．つまり，`assert` 文が実行されるときには引数が 0 ではないことを，プログラムを解析することによっていわば「証明」することで，アサーションが必ず成り立つことを保証するわけである．

ここでは例として `assert` が成り立っていることを保証するための解析について取り上げたがより一般に「プログラムが意図（＝仕様）通りに動作することを証明するための静的解析」を**形式検証** (*formal verification*) と呼ぶ.[3]

形式検証は，テストを補完する方法として最近結構使われ始めている.[4]本章は，簡単な形式検証手法を実装してみることにより，静的解析に馴染んでもらうことを目的としている．静的解析のうち，上で取り上げた最適化については，後日講義で取り上げる予定である．

本章で実装する形式検証手法は**静的な型推論** (*static type inference*) である．OCaml でプログラムを書くと自動で型を推論してくれて，型エラーがあれば知らせてくれるアレである．型推論はプログラム中の式が（評価が停止するならば）どのような値を返すかを**プログラムを実行することなく**解析するので静的解析の一種である．また，プログラムが**実行時に型エラーを起こさない**ことを証明するための手法であるため，形式検証と言える.[5]

---

[3]ちなみに，上記のプログラムでは，`while` ループで条件がテストされる際に必ず `ret - i == x && i <= y` が成り立っていることを発見できれば，アサーションが必ず成り立つことが証明できる.（ループを抜けたときには，`while` 文の条件節が偽になるはずなので，`i >= y` が成り立っているはずである．すると，上記の条件と合わせて `ret - i == x && i == y` が成り立っていることになり，ここから `ret == x + y` が導ける.）このようなループ文の先頭に到達したときに必ず成り立っている条件を**ループ不変条件** (*loop invariant*) と呼ぶ．良いループ不変条件を発見するのは，形式検証においてとても重要なテクニックである．また，自分でプログラムを書く際にも，ループ不変条件を意識して書くことで，バグを減らせることが多い．

[4]例えば Facebook は infer という形式検証ツールをソースコード管理ツールと統合して動作させており，プログラマがコミットした内容を自動で検証し，誤りの可能性を自動的に指摘するということをやっているとのことである [].

[5]上で説明したちょっと格好いい形式検証に比べて，だいぶ保証する性質がしょぼく見えるかも知れないが，人間はそういうしょぼいエラーを含むプログラムを頻繁に書く（実行時型エラーに遭遇してしょんぼりした経験ない？），軽い解析なのに結構役に立つ，モジュールに基づく情報の隠蔽と相性が良い，関数型言語ととても相性がよい，型推論をベースにしてさらに格好いい形式検証を作ることができる等の利点がある．

本章では，3 章で実装した言語のための型推論を解説する．まず ML^2 のための型推論からはじめて，徐々に言語を拡張しつつ，拡張された言語機能のための型推論を行う方法を解説する．

4.2 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) 以下のようでった．

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

ここでは ⟨ 式 ⟩ の代わりに e という記号 (メタ変数)，⟨ 識別子 ⟩ 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，（ML^2 は関数を含まず，値は整数値とブール値のみなので）整数を表す `int`, 真偽値を表す `bool` を考える．[6]

$$\tau ::= \texttt{int} \mid \texttt{bool}.$$

4.2.1 型判断と型付け規則

我々がこれから作ろうとする型推論アルゴリズムは，式 e を受け取って，その e の型を（くどいようだが）e を評価することなく推論する．ここでさらっと「e の型」と書いたが，この言葉の意味するところはそんなに明らかではない．素直に考えれば「e を評価して得られる値 v の型」ということになるのだが「じゃあ v の型って何？」「v の型を定義できたとして，型推論アルゴリズムが正しくその型を推論できていることはどう保証するの？」「e が停止しないかもしれないプログラムだったら評価して得られる値はどう定義するの？」などの問題点にクリアに答えられるようにアルゴリズムを作りたい．

そのために，型推論アルゴリズムを作る際には，普通**型とは何か，プログラム e が型 τ を持つのはどのようなときか**等をまず厳密にに定義し，その型を発見するためのアルゴリズムとして型推論アルゴリズムを定義することが多い．このような，型に関する定義やアルゴリズムを含む体系を**型システム** (*type system*) と呼ぶ.[7]具体的には,「式 e が型 τ を持つ」とい

[6]**メタ変数** (*metavariable*) とは，プログラム中で使われる普通の変数と異なり,「式」「値」「型」などのプログラム中で現れる「もの」を総称的に指すために使われる変数である．例えば，上記の BNF では「式」を表すメタ変数として e が,「自然数」を表すメタ変数として n が用いられている．また，少しややこしいが,「変数」を表すメタ変数として x が用いられている．なお,「式 (expression) を表すメタ変数として e を用いる」ことを表す英語の表現はいくつかあり，"Expressions are ranged over by metavariable e" とか，"e is the metavariable that represents an expression" とか言ったりする．

[7]大体動けばいいんだよ，こまけぇこたぁいいんだよ！という考えもあるだろうが，だいたい動くと思って作ったものが動かないことはよくある．

う関係を**型判断** (*type judgment*) と呼び，$e:\tau$ と略記する.[8]何が正しい型判断で，何が間違った型判断なのかをあとで定義するのだが，例えば「式 $1+1$ は `int` を持つ」ように型システムを作りたいので，$1+1:$ `int` は正しい型判断になるように，式 if 1 then $2+3$ else $4:$ `int` は型がつかないプログラムなので，if 1 then $2+3$ else $4:\tau$ はいかなる τ についても正しくない型判断となるようにしたい.

しかし，型判断を定義するのに e と τ だけでは実は情報が足りない．一般に式には自由変数が現れるからである.，例えば「式 $x+2$ は `int` を持つ」は正しい判断にしたいだろうか.「それは x の型による」としか言いようがない.（x が `int` 型であれば正しい判断にしたいし，x が `bool` 型であれば正しい型判断と認めたくはないだろう.）このため，自由変数を含む式に対しては，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) と呼ぶ．型環境は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

$\Gamma(x) =$ `int` の時 $x:$ `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．また，空の型環境を $\emptyset$ で表す．$\vdash$ 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考えてもよい．式 $\Gamma \vdash e:\tau$ が成り立つような Γ と τ が存在するときに，e に**型がつく** (*well-typed*)，あるいは e は**型付け可能** (*typable*) という．逆にそのような Γ と τ が存在しないときに，e は**型がつかない** (*ill-typed*)，あるいは e は**型付け不能** (*untypable*) という.

型環境の表し方 $\Gamma(x_1)=\tau_1,\ldots,\Gamma(x_n)=\tau_n$ を満たし，それ以外の変数については型が定義されていないような型判断を $x_1\!:\!\tau_1,\ldots,x_n\!:\!\tau_n$ と書くことが多い.

型判断を導入したからには「正しい型判断」を定義しなければならない．これには**型付け規則** (*typing rule*) を使うのが定石である．これは，記号論理学の証明規則に似た「正しい型判断」の導出規則で

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

という形をしている．横線の上の $\langle\,\text{型判断}_1\rangle,\ldots,\langle\,\text{型判断}_n\rangle$ を規則の**前提** (*premise*)，下にある $\langle\,\text{型判断}\,\rangle$ 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.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1{+}e_2 : \texttt{int}} \qquad (\text{T-P}\textsc{lus})$$

この，型付け規則の直感的な意味（読み方）は,

[8]「判断」という言葉はちょっと奇異に感じられるかもしれないが，そういうもんだと思ってほしい．今 Twitter で語源がどこにあるか聞いて回っているところである.

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.[9]

以下に，ML^2 の型付け規則を示す.

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

$$\frac{}{\Gamma \vdash n : \texttt{int}} \tag{T-Int}$$

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

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{+}e_2 : \texttt{int}} \tag{T-Plus}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{*}e_2 : \texttt{int}} \tag{T-Mult}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{<}e_2 : \texttt{bool}} \tag{T-Lt}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 : \tau} \tag{T-If}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textsf{let } x = e_1 \textsf{ in } e_2 : \tau_2} \tag{T-Let}$$

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，

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

[9]（この脚注は意味がわからなければ飛ばして良い.）厳密には T-Plus はメタ変数 e_1, e_2, Γ を具体的な式や型環境に置き換えて得られる（無限個の）導出規則の集合を表したものである．例えば，$\emptyset \vdash 1 : \texttt{int}$ という型判断が既に導出されていたとしよう．T-Plus 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*)

$$\frac{\emptyset \vdash 1 : \texttt{int} \qquad \emptyset \vdash 1 : \texttt{int}}{\emptyset \vdash 1 + 1 : \texttt{int}}$$

が得られる．この具体化された規則を使うと，型判断 $\emptyset \vdash 1 + 1 : \texttt{int}$ が導出できる．

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．

各々の型付け規則がなぜそのように定義されているか，少しずつ説明を加える．[10]

T-VAR: $\Gamma(x) = \tau$ であれば，Γ のもとで式 x が型 τ を持つという判断を導出してよい．Γ が式の中の自由変数の型を決めているという上述の説明から理解できるはずである．

T-INT,T-BOOL: 整数定数 n は，いかなる型環境の下でも型 `int` を持つ．また，式 `true` と式 `false` は，いかなる型環境の下でも型 `bool` を持つ．これらは直観的に理解できると思う．

T-PLUS,T-MULT: 型環境 Γ の下で式 e_1 と式 e_2 が型 `int` を持つことが導出できたならば，Γ の下で式 $e_1{+}e_2$ が `int` を持つことを導出してよい．式 $e_1{*}e_2$ も同様である．これらは式 $e_1{+}e_2$ と $e_1{*}e_2$ が，それぞれ整数の上の演算であることから設けられた規則である．

T-LT: 型環境 Γ の下で式 e_1 と式 e_2 が型 `int` を持つことが導出できたならば，Γ の下で式 $e_1{<}e_2$ が `bool` を持つことを導出してよい．これらは式 $e_1{<}e_2$ が整数の比較演算で，返り値がブール値であることから設けられた規則である．

T-IF: 型環境 Γ の下で式 e_1 が `bool` を持ち，式 e_2 と式 e_3 が**同一の**型 τ を持つならば，if e_1 then e_2 else e_3 がその型 τ を持つことを導出してよい．式 e_1 は if 式の条件部分なので，型 `bool` を持つべきであることは良いであろう．式 e_2 と式 e_3 が同一の型 τ を持つべきとされていること，if 式全体としてその型 τ を持つとされていることについては少し注意が必要である．これは，条件式 e_1 が true と false のどちらに評価されても実行時型エラーが起こらないようにするために設けられている条件である．これにより，**実際は絶対に実行時型エラーが起こらないのに型付け可能ではないプログラムが生じる**．たとえば，(if true then 1 else false) + 3 というプログラムを考えてみよう．このプログラムは，if 式が必ず 1 に評価されるため，実行時型エラーは起こらない．しかし，この if 式の then 節の式 1 には型 `int` がつき，else 節の式 false には型 `bool` がつくので，if 式は型付け不能である．[11]

T-LET: 型環境 Γ の下で式 e_1 が型 τ_1 を持ち，式 e_2 が Γ を $x \mapsto \tau_1$ というエントリでえ拡張して得られる型環境 $\Gamma, x\colon\tau_1$ の下で型 τ_2 を持つならば，式 let $x = e_1$ in e_2 は全体として τ_2 を持つという判断を導いてよい．この規則は let 式がどのように評価されるかと合わせて考えると分かりやすい．式 let $x = e_1$ in e_2 を評価する際には，まず e_1 を現在の環境で評価し，得られた結果に x を束縛した上で e_2 を評価して，その結果を全体の評価結果とする．そのため，型付け規則においても，e_1 の型付けには（「現在の環境」に

[10] 一応書いておくと，ここで説明するのはあくまで理解の助けにするための，型付け規則の背後にある直観であって，型付け規則自体ではない．

[11] ある式 e が型付け不能であることを言うには，**いかなる Γ と τ をもってきても**，$\Gamma \vdash e\colon\tau$ を導けないことを言わなければならないので，この説明は厳密には不十分である．

対応する）型環境 Γ を使い，e_2 の型付けには e_1 の型 τ_1 を x の型とした型環境 $\Gamma, x : \tau_1$ を用いるのである．

ここで型判断 $\Gamma \vdash e : \tau$ が**導出できる** derivable とは，根が型判断 $\Gamma \vdash e : \tau$ で，上記のすべての辺が型付け規則に沿っている木が存在することである．（すべての葉は前提が無い型付け規則が適用された形になっている．）この木を型判断 $\Gamma \vdash e : \tau$ を導出する**導出木** (*derivation tree*) という．例えば，以下は型判断 $x : \mathtt{int} \vdash \mathsf{let}\ \mathsf{y} = 3\ \mathsf{in}\ \mathsf{x} + \mathsf{y} : \mathtt{int}$ の導出木である．

$$
\dfrac{\dfrac{}{x : \mathtt{int} \vdash 3 : \mathtt{int}}\ \textsc{T-Int} \qquad \dfrac{\dfrac{}{x : \mathtt{int}, y : \vdash x : \mathtt{int}}\ \textsc{T-Var} \qquad \dfrac{}{x : \mathtt{int}, y : \vdash y : \mathtt{int}}\ \textsc{T-Var}}{x : \mathtt{int}, y : \mathtt{int} \vdash x + y : \mathtt{int}}\ \textsc{T-Plus}}{x : \mathtt{int} \vdash \mathsf{let}\ y = 3\ \mathsf{in} x + y : \mathtt{int}}\ \textsc{T-Let}
$$

この導出木が存在することが，型判断 $x : \mathtt{int} \vdash \mathsf{let}\ \mathsf{y} = 3\ \mathsf{in}\ \mathsf{x} + \mathsf{y} : \mathtt{int}$ が正しいということの定義である．

4.2.2 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．もしそのような型がなければエラーを報告する．

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だろうか．これは，$\Gamma \vdash e : \tau$ を根とする導出木を構築すればよい．では，このような導出木をどのように作ればよいだろうか．

この答えは型付け規則から得られる．上に挙げた型付け規則は**構文主導な規則** (*syntax-directed rules*) になっているというよい性質を持っている．これは，Γ と e が与えられたときに，$\Gamma \vdash e : \tau$ が成り立つような τ が存在するならば，これを導くような規則が e の形から一意に定まるという性質である．例えば，Γ と e が与えられ，e が $e_1 + e_2$ という形をしていたとしよう．このとき，型推論アルゴリズムは $\Gamma \vdash e : \tau$ を根とする導出木を構築しようとする．型付け規則をよく見ると，このような導出木は（存在するならば）最後の導出規則が $\textsc{T-Plus}$ でしかありえない．すなわち，

$$
\dfrac{\vdots}{\Gamma \vdash e : \tau}\ \textsc{T-Plus}
$$

という形の導出木だけを探索すればよいことになる．このように適用可能な最後の導出規則が e の形から一意に定まる型付け規則を構文主導であるという．

構文主導な型付け規則を持つ型システムでは，各規則を下から上に読むことによって型推論アルゴリズムを得ることができることが多い．例えば，$\textsc{T-Int}$ は入力式が整数リテラルならば，型環境に関わらず，$\mathtt{int}$ を出力する，と読むことができる．また，$\textsc{T-Plus}$ は

入力式 e が $e_1 + e_2$ の形をしていたならば，Γ と e_1 を再帰的に型推論アルゴリズムに入力して型を求めて（これを τ_1 とする）Γ と e_2 とを再帰的に型推論アルゴリズムに入力して型を求めて（これを τ_2 とする）τ_1 も τ_2 も両方とも `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.[12]

Exercise 4.2.1 図 *4.1*, 図 *4.2* 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.

4.3　ML^3 の型推論

4.3.1　関数に関する型付け規則

次に，fun 式，関数適用式

$$e ::= \cdots \mid \textsf{fun}\ x \rightarrow e \mid e_1\ e_2$$

で型推論アルゴリズムを拡張しよう.「τ_1 の値を受け取って（計算が停止すれば）τ_2 の値を返す関数」の型を $\tau_1{\rightarrow}\tau_2$ とすると，型の定義は以下のように変更される.

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1{\rightarrow}\tau_2$$

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.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \textsf{fun}\ x \rightarrow e : \tau_1{\rightarrow}\tau_2} \qquad \text{(T-ABS)}$$

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

それぞれの規則について簡単に説明を加える.

T-ABS: 引数 x が τ_1 を持つという仮定の下で関数本体 e が τ_2 型を持つならば，fun $x \rightarrow e$ が $\tau_1{\rightarrow}\tau_2$ 型を持つことを導いて良い．これは関数のセマンティクスから理解できるであろう.

T-APP: e_1 の型が関数型 $\tau_1{\rightarrow}\tau_2$ であり，かつ，その引数の型 τ_1 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 τ_2 がつくことを導いて良い．これも関数型の直観と関数適用のセマンティクスから分かるであろう.

[12] 明示的に導出木を構築していないので，なぜこれで「導出木を構築している」ことになるのかよくわからないかもしれない．この型推論アルゴリズムは再帰呼出しをしているが，この再帰呼出しの構造が導出木に対応している.

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 4.1: 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 4.2: ML^2 型推論の実装 (2)

次は型推論アルゴリズムの設計である．これらの規則を含めても型付け規則は構文主導なので，前節の「規則を下から上に読む」という戦略を使ってみよう．入力として型環境 Γ と式 e が与えられ，式 e が fun $x \to e_1$ という形をしていたとしよう．そうすると，T-Fun を下から上に使うことに読んで，以下のように型推論ができそうである．

1. 型環境 $\Gamma, x : \tau_1$ と式 e_1 を入力として型推論アルゴリズムを再帰的に呼び出し型 τ_2 を得る．

2. 型 $\tau_1 \to \tau_2$ を e の型として返す．

ところが，これでは型推論が実装できない．問題は，最初のステップで e_1 の型を調べる際に作る型環境 $\Gamma, x : \tau_1$ である．ここで x の型として τ_1 を取っているが，この型をどのように取るべきかは，一般には e_1 の中での x の使われ方と，この関数 fun $x \to e_1$ がどのように使われうるかに依存するので，このタイミングで τ_1 を容易に決めることはできない．

簡単な例として，fun x $\to$ x $+$ 1 という式を考えてみよう．これは，`int`$\to$`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 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．

4.3.2 型変数，型代入と型推論アルゴリズムの仕様

この「τ_1 の適切な取り方が後にならないとわからない」という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*) を導入しよう．

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

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，x $+$ 1 の型は `int` である，という情報に加え $\alpha = \texttt{int}$ 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 $\alpha \to \texttt{int}$，つまり，`int`$\to$`int` であることがわかる．また，fun x $\to$ fun y $\to$ 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. 新しい（つまり，他の型変数とカブらない）型変数 α を生成し，x の型を α と置いて，本体，つまり fun y $\to$ x y の型推論を行う．

2. 新しい型変数 β を生成し，y の型を β と置いて，本体，つまり x y の型推論を行う．

3. x y の型推論の結果，この式の型が別の新しい型変数 γ を使って $\beta \to \gamma$ と書け，$\alpha = \beta \to \gamma$ であることが判明する．

さらに詳しい型推論アルゴリズムの中身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.[13]

> **型変数と多相型** OCaml の多相型 (*polymorphic type*) とここで導入する型変数を含む型とを混同してはならない．OCaml における多相型は，例えば OCaml では `fun x -> x` が型 `'a -> 'a` を持ち，ここで表示される `'a` を「型変数」ということがあるが，この `'a` は上記の型変数とは異なる．この `'a` は「何の型にでも置き換えてよい」変数であるが，上記の型変数は「特定の型を表す」記号である.「任意の型で置き換え可能」な型変数は次節で再登場する.

ここまで述べたことを実装したのが図 4.3 である．型 `ty` を型変数を表すコンストラクタ `TyVar` と関数型を表すコンストラクタ `TyFun` とで拡張する．`TyVar` は `tyvar` 型の値を一つとるコンストラクタで `TyVar(tv)` という形をしており，これが型変数を表す．`tyvar` 型は型変数の名前を表す型で，実体は整数型である．`TyFun` は `ty` 型の引数を 2 つ持つ `TyFun(t1,t2)` という形をしており，これが型 $\tau_1 \rightarrow \tau_2$ を表す.

型推論アルゴリズムの実行中には，他のどの型変数ともかぶらない新しい型変数を生成する必要がある.（このような型変数を ***fresh* な型変数**と呼ぶ.）これを行うのが関数 `fresh_tyvar` である．この関数は引数として `()` を渡すと（すなわち `fresh_tyvar ()` のように呼び出すと）新しい未使用の型変数を生成する．この関数は次に生成すべき型変数の名前を表す整数への参照 `counter` を保持しており，保持している値を新しい型変数として返し，同時にその参照をインクリメントする．上で説明したように，T-Abs のケースでは新しい型変数を生成するのだが，その際にこの関数を使用する.[14]

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（定義域が有限集合な）写像である．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．例えば $\mathcal{S}$ が $\{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{bool}\}$ であるとき，$\mathcal{S}\alpha = \texttt{int}$ であり，$\mathcal{S}(\alpha \rightarrow \beta) = \texttt{int} \rightarrow \texttt{bool}$ であり，$\mathcal{S}(x:\alpha, y:\beta) = (x:\texttt{int}, y:\texttt{bool})$ である．$\mathcal{S}\tau$，$\mathcal{S}\Gamma$

---

[13] 余談ではあるが，ここで用いられている方法は，未知の情報を含む問題を解くために (1) 未知の情報をとりあえず変数において (2) その変数が満たすべき制約を生成し (3) その制約を解くという，より一般的な問題解決の手法の適用例と見ることができる．方程式を立ててそれを解くとか，散々やってきたよね？

[14] 関数 `fresh_tyvar` は呼び出すたびに異なる値を返すことに注意せよ．これは `fresh_tryvar` が純粋な意味での計算ではない（参照の値の更新や参照からの値の呼び出しといった）**副作用** (*side effect*) を持つためである.

はより厳密には以下のように定義される．

$$
\begin{array}{rcl}
\mathcal{S}\alpha & = & \begin{cases} \mathcal{S}(\alpha) & \text{if } \alpha \in \mathit{dom}(\mathcal{S}) \\ \alpha & \text{otherwise} \end{cases} \\
\mathcal{S}\texttt{int} & = & \texttt{int} \\
\mathcal{S}\texttt{bool} & = & \texttt{bool} \\
\mathcal{S}(\tau_1 \rightarrow \tau_2) & = & \mathcal{S}\tau_1 \rightarrow \mathcal{S}\tau_2 \\
 & & \\
\mathit{dom}(\mathcal{S}\Gamma) & = & \mathit{dom}(\Gamma) \\
(\mathcal{S}\Gamma)(x) & = & \mathcal{S}(\Gamma(x))
\end{array}
$$

$\mathcal{S}\alpha$ のケースが実質的な代入を行っているケースである．$\mathcal{S}$ の定義域 $\mathit{dom}(\mathcal{S})$ に α が入っている場合は，$\mathcal{S}$ によって定められた型（すなわち $\mathcal{S}\alpha$）に写像する．int と bool は型変数を含まないので，$\mathcal{S}$ を適用しても型に変化はない．τ が $\tau_1 \rightarrow \tau_2$ であった場合は再帰的に $\mathcal{S}$ を適用する．

型代入を使うと，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．

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 $\mathcal{S}$

型推論アルゴリズムを実装する前に，以降で使う補助関数を定義しておこう．

Exercise 4.3.1 図 *4.3* 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 MySet.t
```

とする．型 'a MySet.t は mySet.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．

さて，型推論アルゴリズムを実装するためには，型代入を表すデータ構造を決める必要がある．様々な表現方法がありうるが，ここでは素直に型変数と型ののペアのリストで表現することにしよう．すなわち，型代入を表す OCaml の型は以下のように宣言された subst である．

```
type subst = (tyvar * ty) list
```

subst 型は [(id1,ty1); ...; (idn,tyn)] の形をしたリストである．このリストは $[\texttt{idn} \mapsto \texttt{tyn}] \circ \cdots \circ [\texttt{id1} \mapsto \texttt{ty1}]$ という型代入を表すものと約束する．つまり，この型代入は「受け取った型中の型変数 id1 をすべて型 ty1 に置き換え，得られた型中の型変数 id2 をすべて型 ty2 に置き換え... 得られた型中の型変数 idn をすべて型 tyn に置き換える」ような代入である．リスト中の型変数と型のペアの順序と，代入としての作用の順序が逆になっているこ

とに注意してほしい．また，リスト中の型は後続のリストが表す型代入の影響を受けることに注意してほしい．例えば，型代入 `[(alpha, TyInt)]` が型 `TyFun(TyVar alpha, TyBool)` に作用すると，`TyFun(TyVar TyInt, TyBool)` となり，型代入

```
[(beta, (TyFun (TyVar alpha, TyInt))); (alpha, TyBool)]
```

が型 `(TyVar beta)` に作用すると，まずリストの先頭の `(beta, (TyFun (TyVar alpha, TyInt)))` が作用して `TyFun (TyVar alpha, TyInt)` が得られ，次にこの型にリストの二番目の要素の `(alpha, TyBool)` が作用して `TyFun(TyBool, TyInt)` が得られる．

以下の演習問題で，型代入を作用させる補助関数を実装しよう．

Exercise 4.3.2 型代入に関する以下の型，関数を `typing.ml` 中に実装せよ．

```
type subst = (tyvar * ty) list

val subst_type : subst -> ty -> ty
```

例えば，

```
let alpha = fresh_tyvar () in
subst_type [(alpha, TyInt)] (TyFun (TyVar alpha, TyBool))
```

の値は `TyFun (TyInt, TyBool)` になり，

```
let alpha = fresh_tyvar () in
let beta = fresh_tyvar () in
subst_type [(beta, (TyFun (TyVar alpha, TyInt))); (alpha, TyBool)] (TyVar beta)
```

の値は `TyFun (TyBool, TyInt)` になる．

4.3.3　単一化

型変数と型代入を導入したところで型付け規則をもう一度見てみよう．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．

これらの制約を ML[2] に対する型推論では，型 (すなわち `TyInt` などの定義される言語の型を表現した値) の比較を行うことでチェックしていた．例えば与えられた式 e が $e_1 + e_2$ の形をしていたときには，e_1 の型 τ_1 と e_2 の型 τ_2 を再帰的にアルゴリズムを呼び出すことにより推論し，**それらが `int` であることをチェックしてから**全体の型として `int` を返していた．

しかし，型の構文が型変数で拡張されたいま，この方法は不十分である．というのは，部分式の型（上記の τ_1 と τ_2）に型変数が含まれるかもしれないからである．例えば，fun x $\rightarrow$ 1 + x という式の型推論過程を考えてみる．まず，$\emptyset \vdash$ fun x $\rightarrow$ 1 + x : `int`$\rightarrow$`int` であることに注意

Makefile:

```
OBJS= mySet.cmo syntax.cmo parser.cmo lexer.cmo \
   environment.cmo typing.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 MySet.t *)
```

図 4.3: ML^3 型推論の実装 (1)

しよう．（実際に導出木を書いてチェックしてみること．）したがって，型推論アルゴリズムは，この式の型として int→int を返すように実装するのが望ましい．

では，空の型環境 $\emptyset$ と上記の式を入力として，型推論アルゴリズムがどのように動くべきかを考えてみよう．この場合，まず T-Fun を下から上に読んで，x の型を型変数 α とおいた型環境 $x\!:\!\alpha$ の下で $1+$x の型推論をすることになる．その後，各部分式 1 と x の型を，アルゴリズムを再帰的に呼び出すことで推論し，int と α を得る．ML² の型推論では，ここでそれぞれの型が int であるかどうかを単純比較によってチェックし，int でなかったら型エラーを報告していた．しかし今回は後者の型が α であって int ではないため，**単純比較による部分式の型のチェックだけでは型推論が上手くいかない．**

では，どうすれば良いのだろうか．定石として知られている手法は**制約による型推論** (*constraint-based type inference*) という手法である．この手法では，与えられたプログラムの各部分式から型変数に関する**制約** (*constraint*) が生成されるものと見て，式をスキャンする過程で制約を集め，その制約をあとで解き型代入を得る，という形で型推論アルゴリズムを設計する．例えば，上記の例では,「未知だった α は実は int である」という制約が生成される．この制約を解くと型代入 $\{\alpha \mapsto \texttt{int}\}$ が得られる．

上記の場合は制約が単純だったが，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，

> 与えられた型のペアの集合 $\{(\tau_{11},\tau_{12}),\ldots,(\tau_{n1},\tau_{n2})\}$ に対して，$\mathcal{S}\tau_{11}=\mathcal{S}\tau_{12}$，$\ldots$，$\mathcal{S}\tau_{n1}=\mathcal{S}\tau_{n2}$ なる $\mathcal{S}$ を求めよ

という制約解消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 と int は $\mathcal{S}(\alpha)=\texttt{int}$ なる型代入 $\mathcal{S}$ により単一化できる．また，$\alpha\to\texttt{bool}$ と $(\texttt{int}\to\beta)\to\beta$ は $\mathcal{S}(\alpha)=\texttt{int}\to\texttt{bool}$ かつ $\mathcal{S}(\beta)=\texttt{bool}$ 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[15]，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*) と呼ばれる問題）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．[16]（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．）

一階の単一化を行うアルゴリズム $\mathcal{U}(X)$ は，型のペアの集合 X を入力とし，X 中のすべての型のペアを同じ型にするような型代入を返す．（そのような型代入が存在しないときには

[15] 問題設定によっては**決定不能** (*undecidable*) になることもある．決定不能であるとは，いい加減に言えば，かつすべての入力について有限時間で停止し正しい出力を返すプログラムが存在しないことを言う．従って，決定不能な問題を計算機でなんとかしようとすると，一部の入力については正しくない答えを返すことを許容するか，一部の入力については停止しないことを許容しなければならない．

[16] このアルゴリズムは，Prolog などの論理型言語と呼ばれるプログラミング言語の処理系において多く用いられる．

エラーを返す.）$\mathcal{U}$ は以下のように定義される.

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

ここで，$\emptyset$ は空の型代入を表し，$[\alpha \mapsto \tau]$ は α を τ に写す（そしてそれ以外の型変数については何も行わない）型代入である．また $\mathit{FTV}(\tau)$ は τ 中に現れる型変数の集合である．また，$X \uplus Y$ は，$X \cap Y = \emptyset$ のときの $X \cup Y$ を表す記号である．

このの $\mathcal{U}$ の定義は以下のように X を入力とする単一化アルゴリズムとして読める:

- X が空集合であれば空の代入を返す．
- そうでなければ，X から型のペア (τ_1,τ_2) を任意に一つ選び，それ以外の部分を X' とし，(τ_1,τ_2) がどのような形をしているかによって，以下の各動作を行う．
  - τ_1 と τ_2 がすでに同じ形であった場合: X' について再帰的に単一化を行い，その結果を返せばよい.（τ_1 と τ_2 はすでに同じ形なので，残りの制約集合 X' の解がそのまま全体の解となる.）
  - 選んだ型のペアがどちらも関数型の形をしていた場合，すなわち τ_1 が $\tau_{11}{\rightarrow}\tau_{12}$ の形をしており，τ_2 が $\tau_{21}{\rightarrow}\tau_{22}$ の形をしていた場合: τ_1 と τ_2 が同じ形となるためには τ_{11} と τ_{21} が同じ形であり，かつ τ_{12} と τ_{22} が同じ形であればよい．これを満たす型代入を求めるために，$\mathcal{U}$ を $\{(\tau_{11},\tau_{21}),(\tau_{12},\tau_{22})\} \cup X'$ を入力として再帰的に呼び出し，帰ってきた結果を全体の結果とする．
  - 選んだ型のペアが型変数と型のペア，すなわち (α,τ) か (τ,α) の形をしていた場合[17]: この場合，型変数 α は τ でなければならないことがわかる．したがって，残りの制約 X' 中の α に τ を代入した制約 $[\alpha \mapsto \tau]X'$ を再帰的に解き，得られた解に α を τ に代入する写像 $[\alpha \mapsto \tau]$ を合成して得られる写像 $\mathcal{U}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau]$ を解として返せばよい．ところが，ここで注意すべきことが一つある．もし τ 中に α が現れていた場合[18]，ここでエラーを検出しなければならない.（なぜなのかを考察する課題を以下に用意している.）

Exercise 4.3.3 上の単一化アルゴリズムを

[17] (α,α) の形だった場合はこのケースではなく，一つ前のケースに当てはまる．

[18] 繰り返しになるが，τ が α 自体であった場合はこのケースには当てはまらない．ここでエラーを報告しなければならないのは，例えば τ が $\alpha{\rightarrow}\alpha$ の場合である．

```
val unify : (ty * ty) list -> subst
```

として実装せよ．

Exercise 4.3.4 単一化アルゴリズムにおいて，$\alpha \notin \mathit{FTV}(\tau)$ という条件はなぜ必要か考察せよ．

4.3.4 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むと，

1. Γ, e_1 を入力として型推論を行い，$\mathcal{S}_1$，τ_1 を得る．
2. Γ, e_2 を入力として型推論を行い，$\mathcal{S}_2$，τ_2 を得る．
3. 型代入 $\mathcal{S}_1, \mathcal{S}_2$ を $\alpha = \tau$ という形の方程式の集まりとみなして，$\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{(\tau_1, \mathtt{int}), (\tau_2, \mathtt{int})\}$ を単一化し，型代入 $\mathcal{S}_3$ を得る．
4. $\mathcal{S}_3$ と int を出力として返す．

となる．部分式の型推論で得られた型代入を方程式とみなして，再び単一化を行うのは，ひとつの部分式から $[\alpha \mapsto \tau_1]$，もうひとつからは $[\alpha \mapsto \tau_2]$ という代入が得られた時に τ_1 と τ_2 の整合性が取れているか（単一化できるか）を検査するためである．

Exercise 4.3.5 他の型付け規則に関しても同様に型推論の手続きを与えよ *(*レポートの一部としてまとめよ*)*．そして，図 *4.4* を参考にして，型推論アルゴリズムの実装を完成させせよ．

Exercise 4.3.6 [$\star\star$] 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．

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

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 4.3.7 [$\star\star$] 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 $\tau\mathtt{list}$ という型を与える．

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

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

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

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:

```
type subst = (tyvar * ty) list

let rec subst_type subst t = ...

(* eqs_of_subst : subst -> (ty * ty) list
   型代入を型の等式集合に変換                  *)
let eqs_of_subst s = ...

(* subst_eqs: subst -> (ty * ty) list -> (ty * ty) list
   型の等式集合に型代入を適用                                 *)
let subst_eqs s eqs = ...

let rec unify l = ...

let ty_prim op ty1 ty2 = match op with
    Plus -> ([(ty1, TyInt); (ty2, TyInt)], TyInt)
  | ...

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 (s1, ty1) = ty_exp tyenv exp1 in
      let (s2, ty2) = ty_exp tyenv exp2 in
      let (eqs3, ty) = ty_prim op ty1 ty2 in
      let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ eqs3 in
      let s3 = unify eqs in (s3, subst_type s3 ty)
  | IfExp (exp1, exp2, exp3) -> ...
  | LetExp (id, exp1, exp2) -> ...
  | FunExp (id, exp) ->
      let domty = TyVar (fresh_tyvar ()) in
      let s, ranty =
        ty_exp (Environment.extend id domty tyenv) exp in
        (s, TyFun (subst_type s domty, ranty))
  | AppExp (exp1, exp2) -> ...
  | _ -> Error.typing ("Not Implemented!")
```

図 4.4: ML^3 型推論の実装 (2)

4.4 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** (*let polymorphism*) を実装する．

本節を理解するためには OCaml の多相型の知識があったほうがよい．以下の二つのプログラムがどのように型付けされるか，あるいはされないかがよく分からないという読者は，OCaml 入門テキストの多相性に関する節，特に let 多相に関する節を復習してから，この先を読まれたい．

- `let id x = x in (id 3, id true)`
- `(fun id -> (id 3, id true)) (fun x -> x)`

4.4.1 多相性と型スキーム

OCaml 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できる関数の型としたり，`'a` を置き換えて `bool->bool` として真偽値に適用する関数の型としたりすることができる．このように，型変数には「今のところ未確定で後で正体が判明する型変数（**単相的** (*monomorphic*) な型変数）」と「どんな型にでも置き換えてよい型変数（**多相的** (*polymorphic*) な型変数）」の二種類があることに注意しよう．

この二種類を区別するために，多相的な型変数は $\forall\alpha.$ で束縛されるものとし，$\forall\alpha$ が型の前に付けられた表現を**型スキーム** (*type scheme*) と呼ぶことにする．より正確には，型 τ の前に有限個の $\forall\alpha$ が付けられた表現 $\forall\alpha_1.\forall\alpha_2.\ldots.\forall\alpha_n.\tau$ を型スキームと呼ぶ.（この型スキームを，型変数の列 $(\alpha_1,\ldots,\alpha_n)$ をベクトル表記を借りて $\vec{\alpha}$ と書くことにして，以下では $\forall\vec{\alpha}.\tau$ と書くことにする.）例えば $\forall\alpha.\alpha\to\alpha$ は型スキームである．

型スキーム $\forall\vec{\alpha}.\tau$ は，型変数の列 $\vec{\alpha}$ に相当する型を受け取って型を返す，いわば型から型への関数のようなものと見ることができる．例えば，型スキーム $\forall\alpha.\alpha\to\alpha$ は，型 `int` を受け取ったら型 `int`→`int` を返し，型 `bool` を受け取ったら型 `bool`→`bool` を返すものと見ることができる．このように見ると，上記のプログラムでは,

- `let` で `f` が束縛された場所で `f` に型スキーム $\forall\alpha.\alpha\to\alpha$ を割り当て,

- `if` の条件節の式 `f true` 中では割り当てられた型スキームに `bool` を与えることで `f` を `bool`→`bool` 型として使い，
- `then` 節の式 `f 2` 中では，割り当てられた型スキームに `int` を与えることで `f` を `int`→`int` 型として使う，

ことで `f` の多相的な振る舞いを捉えることができる．

より形式的には，型 τ と型スキーム σ の定義を以下のように変更する．

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

新しく導入された型スキーム σ が（上の説明の通り）型 τ の前に有限個の $\forall\alpha$ がついた形になっていることを確認されたい．また，型 τ は型スキームともみなせることに注意されたい．（$\forall\alpha.$ がひとつもついていない型スキームである．）型スキーム中，$\forall$ のついている型変数を**束縛されている** (*bound*) といい，束縛されていない型変数（これらは単相的な型変数である）を**自由である** (*free*)，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．

その上で，型環境 Γ を（変数から型への部分関数ではなく）変数から型スキームへの部分関数とする．これにより，`let` で束縛された変数には型スキーム $\forall\vec{\alpha}.\tau$ を持たせておき，使用する際に $\vec{\alpha}$ を適切な型で置き換えることで，多相的な振る舞いを型システムの上で表現することができる．なお，

> **型と型スキームの区別** ここまでの説明から分かるように，これから導入する型システムでは型と型スキームを区別する．この区別は，技術的には，型に相当するメタ変数 τ と型スキームに相当するメタ変数 σ を区別していることから生じており，この区別のために $(\forall\alpha.\alpha)\to(\forall\alpha.\alpha)$ のような表現は型とはみなされないようになっている．
>
> 　型と型スキームを区別して型システムを設計するのは，主に型推論問題の決定可能性の要請から来ている．
>
> $$\tau \quad ::= \quad \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau.$$
>
> のように型スキームも型とみなせるように型を定義して型システムを設計すると，より多くのプログラムを型付け可能とすることができ，型システムの表現力は上がるのだが，素朴な同一視をするだけでは型推論問題が決定不能になることが知られている．型と型スキームを区別し（あとで見るように）多相性のある変数を導入できる場所を `let` や `let rec` に制限することで，実行時型エラーを含まない十分に多くのプログラムを型付け可能とすることができ，なおかつ型推論問題を決定可能とすることが可能となる．

図 4.5 上半分に型スキームの実装上の定義を示す．関数 `freevar_tysc` は，Exercise 4.3.1 で実装した関数 `freevar_ty` の拡張で，型スキーム σ を受け取り，σ に自由に出現する型変数の集合を計算する関数である．$\forall\vec{\alpha}$ が型変数列 $\vec{\alpha}$ を束縛するため，型スキーム $\forall\vec{\alpha}.\tau$ 中に出現する自由な型変数の集合は，型 τ 中に出現する型変数の集合から $\vec{\alpha}$ 中の型変数をすべて除いたものになる．

4.4.2　型付け規則の拡張

変数のための規則

次に，型付け規則をどのように拡張すればよいのか考えてみよう．型環境が変数から型スキームへの部分関数であることを思い出されたい．変数のための規則は以下の通りになっている．

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

すなわち，型環境中で x が束縛されている先の型スキームを $\forall\alpha_1, \ldots, \alpha_n.\tau$ とすると，τ 中の $\alpha_1, \ldots, \alpha_n$ を任意の型 $\tau_1, \ldots, \tau_n$ で置き換えて得られる型を x の型としてよい．（まさにこれがやりたかったことである．）例えば，$\Gamma(\mathsf{f}) = \forall\alpha.\alpha\to\alpha$ とすると，

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

や

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

といった型判断を導出することができる．[19]

let (rec) 式に関する規則

さて，let に関しては，大まかには以下のような規則になるはずである．

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad (\textsc{T-PolyLet?})$$

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．例えば，

let f x = ((let g y = (x, y) in g 4), x + 1) in ...

という式を考え，その型推論の経過を書くと，

1. x の型を α とし，式 `((let g y = (x, y) in g 4), x + 1)` の型推論をする．

[19] なお，あとで定義する型推論アルゴリズムは，プログラム全体に型が付くように適切な $\tau_1, \ldots, \tau_n$ を選ばなければならない．しかしながら，型付け規則を定義する段階においては，適切な $\tau_1, \ldots, \tau_n$ は何なのかを気にする必要はない．このように，型付け規則は数学的にきれいな形で書いておいて，型推論アルゴリズムで適切な $\tau_1, \ldots, \tau_n$ を選ぶ等の作業を頑張るといった役割分担は，型システム関係の文献では頻出である．

2. 第 1 要素の式 `let g y = (x, y) in g 4` の型推論を行う．このために，関数 `g` のパラメータ `y` の型を β とし，型推論を行う．関数の型として $\beta \to \alpha * \beta$ が得られる．

ここで，`g` は `let` で束縛されているので，推論された型 $\beta \to \alpha * \beta$ から型スキームを作り，g をこの型スキームに束縛する必要がある．ここで $\forall$ で束縛してよい（つまり多相的に使って良い）型変数はどれであろうか．

もし $\forall \alpha.\forall \beta.\beta \to \alpha * \beta$ のように α についてまで束縛して（$\forall$ をつけて）しまったとしよう．すると，関数 `f` は型 $\forall \alpha.\forall \beta.\beta \to \alpha * \beta$ を持つことになる．これは `f` に対してどのような型の引数でも渡せることを意味するから，`f true` といった式が `in` の後に書かれていても許されることになる．しかし，`f` の定義中に `x + 1` という式が現れているため，これでは `true + 1` を実行中に評価することになってしまい，実行時に型エラーが起こってしまう．

何がおかしかったのだろうか．α が `g` のスコープの外側で宣言されている `x` の型であったことである．スコープの外側で宣言されている変数の型は，その変数が外側でどのように使われているかに依存して決まるため，後になって特定の型にしなければならない場合がある．（実際にこの例では `x` が外側で `x+1` のように整数との加算に用いられているため，`x` の型を `int` としなければならないことが，後になって分かる．）そのため，$\forall$ をつけて多相性を持たせてはならないのである．というわけで，正しい型付け規則は，付帯条件をつけて，

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

となる.「Γ に自由に出現しない」という条件で，スコープ外で宣言された変数の型として使われている型変数が多相性を持たないように制限している．

4.4.3 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，実は型推論の実装に対する変更はそんなに多くはない．メジャーな変更が必要なのは変数式に関するケースと let 式に関するケースである．図 4.6 にコードの変更点を示す．

まず，変数式に関するケースを考えよう．型変数に代入する型（型付け規則中の $\tau_1, \ldots, \tau_n$）はこの時点では未知であり，変数が他の部分でどう使われるかに依存して決定される．そのため，ここでは $\tau_1, \ldots, \tau_n$ に相当する新しい型変数を用意し，それらをつかって具体化を行う．

次に let 式のケースである．ここでは，e_1 の型推論で得られた e_1 の型 τ を型スキーム化する必要がある．型スキームする際には，T-PolyLet の付帯条件を満たすように多相性を持たせる型変数を決定する必要がある．この計算を行うための補助関数として図 4.6 中で `closure` を定義している．これは，型 τ と型環境 Γ と型代入 S から，条件「$\alpha_1, \ldots, \alpha_n$ は τ に自由に出現する型変数で $S\Gamma$ には自由に出現しない」を満たす型スキーム $\forall \alpha_1.\cdots.\forall \alpha_n.\tau$ を求める関数である．型代入 S を引数に取るのは，型推論の実装に便利なためである．

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 4.5: 多相的 let のための型推論の実装 (1)

Exercise 4.4.1 [$\star\star$] 図 4.5, 4.6 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 4.4.2 [$\star$] 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．

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

Exercise 4.4.3 [$\star\star\star$] OCaml 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 4.4.4 [$\star\star\star$] 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．

typing.ml

```
type tyenv = tysc Environment.t

let rec freevar_tyenv tyenv = ...

let closure ty tyenv subst =
  let fv_tyenv' = freevar_tyenv tyenv in
  let fv_tyenv =
    MySet.bigunion
      (MySet.map
          (fun id -> freevar_ty (subst_type subst (TyVar id)))
          fv_tyenv') in
  let ids = MySet.diff (freevar_ty ty) fv_tyenv in
    TyScheme (MySet.to_list ids, ty)

let rec subst_type subst = ...

let rec ty_exp tyenv = function
     Var x ->
      (try
        let TyScheme (vars, ty) = Environment.lookup x tyenv in
        let s = List.map (fun id -> (id, TyVar (fresh_tyvar ())))
                  vars in
          ([], subst_type s ty)
       with Environment.Not_bound -> err ("variable not bound: " ^ x))
   | ...
   | LetExp (id, exp1, exp2) -> ...

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

図 4.6: 多相的 let のための型推論の実装 (2)

第5章　ML^{4}コンパイラの設計と実装

コンパイラを書いて疲労困憊ら〜.
名も無き賢人

5.1　能書き

本章ではML^{4}言語からMIPSアセンブリへのコンパイラの設計について解説する．コンパイラは，2章で言及した通り，ソース言語のプログラムを同じ振る舞いをするターゲット言語のプログラムに変換するソフトウェアである．本章で設計するコンパイラでは，ソース言語がML^4，ターゲット言語がMIPSアセンブリということになる．生成されたMIPSアセンブリを世の中にあるアセンブラで実行可能バイナリにさらに変換することにより，ML^{4}プログラムをMIPSアーキテクチャの計算機やシミュレータで動かすことが可能になる.[1]

一般的にコンパイラはソースプログラムを一度にターゲット言語に変換するのではなく，その間に幾つかの言語を挟んで，徐々にターゲット言語への変換を行う．間に挟まれるこれらの言語を**中間言語** (*intermediate language*) と呼ぶ．このような設計の利点は

- 徐々に中間言語の抽象度を下げることができ，各変換がわかりやすくなる.[2]
- 新しい言語を設計したときに，中間言語を再利用することができる．すなわち，中間言語Iを作り，Iからアセンブリへの変換を作ってしまえば，将来別のプログラミング言語Lのコンパイラを作る際に，コンパイラ全体を実装する必要はなく，LからIへの変換を実装するだけでよい.

等がある.

本章で設計するコンパイラでは，二つの中間言語を置く．一つ目はML^4 プログラムで明示されていない式の評価順序等の情報を明示した関数型言語，もう一つはMIPSアセンブリにより近い命令形言語である．名前があったほうが教科書を書きやすいので，前者を言語$\mathcal{C}$，後者を言語$\mathcal{V}$と呼ぶことにしよう．また，ターゲット言語であるMIPSアセンブリを言語$\mathcal{A}$と呼ぶことにする．すると，本章で作るコンパイラの概略は図 ??に示すとおりとなる.[3] ⟵

[1]本章ではMIPSアセンブリの知識を（できるだけ）仮定せずに読めるように書いたつもりである．コンパイラの最後のフェーズではさすがにMIPSアセンブリの知識が必要になるんだけれども.

[2]ここで抽象度とは，言語機能のリッチさと思ってもらえればよい．高階関数やオブジェクト指向や○○指向やらがたくさん入った言語で書かれたプログラムを一気にアセンブリに落とすよりは，徐々にアセンブリに

5.1.1 ソース言語

中身の説明に入る前に，ソース言語を再確認しよう．ソース言語は 3 章で定義した ML4 のうち，関数定義はトップレベルのみで行うことに制限した言語である．[4]構文は以下のように定義される．

$$
\begin{array}{rcl}
P & ::= & e;; \mid \textsf{let}\ x{=}e;; \mid \textsf{let rec}\ f = \textsf{fun}\ x \rightarrow e;; \\
e & ::= & x \mid n \mid \textsf{true} \mid \textsf{false} \mid e_1\ op\ e_2 \mid \textsf{if}\ e\ \textsf{then}\ e_1\ \textsf{else}\ e_2 \mid \textsf{let}\ x = e_1\ \textsf{in}\ e_2 \\
 & \mid & \textsf{fun}\ x \rightarrow e \mid e_1\ e_2 \mid \textsf{let rec}\ f = \textsf{fun}\ x \rightarrow e_1\ \textsf{in}\ e_2 \\
op & ::= & + \mid * \mid <
\end{array}
$$

ここで，P，e，op はそれぞれプログラム，式，二項演算子を表す**メタ変数** (*metavariable*) とする．また，x, y, f, g を変数を表すメタ変数とする．メタ変数とは，言語のある要素を表すものとしてあらかじめ定められた記号である．例えば上記の BNF では P は「プログラム」を表すメタ変数として，e は「式」を表すメタ変数として定めておくわけである．このようにすると，いちいち「e_1 と e_2 はそれぞれ式であり...」と断ることなく，単に式 $e_1 + e_2$ と書いただけである形の式を表すことができるわけである．

メタ変数と変数 「メタ」とはギリシア語に語源を持つ接頭語である．プログラミング言語の文脈では「メタ○○」で「対象に言及するための○○」を表すことが多い．例えば「e は式を表すメタ変数である」とは「e は式に言及するための変数である」という意味である．

プログラム中の変数を表すメタ変数 x と変数 x の違いに注意すること．let 式は let $x = e_1$ in e_2 の形をしているが，ここでの x はメタ変数であるから，let x = true in false も let y = true in false もこの形に当てはまることになる．もし e の定義が

$$
\begin{array}{rcl}
\ldots & & \\
e & ::= & \cdots \mid \textsf{let x} = e_1\ \textsf{in}\ e_2 \mid \ldots \\
\ldots & &
\end{array}
$$

と，let 式の x の部分がメタ変数でないプログラム変数の x と書いてあったとすると，後者の let 式は（定義しようとしている変数が x ではないので）式の構文には当てはまらないことになる．

近づけていく方が変換が分かりやすいし実装も容易ということである．

[3]なお，本章で解説するコンパイラは東北大学の住井英二郎氏の MinCaml コンパイラ [?] からつまみ食いをしたものになっている．MinCaml は OCaml さえ読めればとても分かりやすいミニコンパイラになっているので，できればそっちも読んでほしい．

[4]この制限は関数閉包を作らなくともプログラムを実行できるために講義の時間の都合上設けている制限である．実行時に関数閉包が作られうるようなプログラムをアセンブリ言語に落とすには，**クロージャ変換** (*closure conversion*) と呼ばれるプログラム変換を途中で行うなどして，関数がトップレベルで定義される形にプログラムを変換する必要がある．この変換を講義中で扱う時間がないので，ソース言語の方を制限しちゃうのである．これではほとんど C 言語と変わらないのであるが，講義ができないとわしが怒られるので仕方ないのである．興味のある者は上記の住井コンパイラ [] を参照のこと．

5.1.2 言語 $\mathcal{C}$

これから ML^4 プログラムをいくつかの中間言語を経由して MIPS アセンブリまで変換する．ML^4 と MIPS アセンブリの大きな差異の一つは，ML^4 においては，式を評価したときにどのような順序でどのような計算が起こるかが必ずしも明示されていない，すなわち式の評価でどのような**制御** (*control*) が行われるかが明示されていないということがある．例えば，式 (fun x -¿ (x + 1) * 2) (3 + 1) を考えよう．この式を評価する際には

1. 式 (fun x -¿ (x + 1) * 2) を評価してクロージャを作り，
2. 式 3+1 を評価して整数値 4 を得て，
3. はじめに作ったクロージャに値 4 を与え，
4. x が 4 という環境の下で x + 1 を評価して整数値 5 を得て，
5. 得られた整数値 5 を 2 に乗じて整数値 10 を得て，
6. 得られた整数値 10 を全体の計算結果として返す

という計算が起こるはずである．

MIPS アセンブリでは，このような計算順序に関する情報（もう少し正確に言えば，各計算ステップで得られた値が次にどのような計算で用いられるのかに関する情報）を逐一指定しなければならない．したがって，ML^4 を MIPS アセンブリに変換する過程では，計算順序に関する情報を明示化する必要がある．元の式が仮に以下のように書いてあれば，計算順序が明示化されている感じがしないだろうか．

```
let f =
   fun x ->
     let t1 = x + 1 in
     let t2 = t1 * 2 in
     t2
 in
 let y = 3 + 1 in
 let z = f y in
   z
```

このプログラムでは，元のプログラム中のすべての部分式の評価結果に let で何らかの変数が束縛されており，かつ各部分式の評価結果がその後どのような計算でどのように用いられるかが明示されている．例えば，関数 f 中では，x+1 の計算結果がその後「2 を掛ける」という計算で用いられ，さらにその結果が f の返り値として返されることが，t1 を見ることで分かる．

言語 $\mathcal{C}$ は，すべての部分式の評価結果に何らかの変数が束縛されることを強制した関数型言語である．文法は以下の通りである．

$$\begin{array}{rcl}
P & ::= & e;; \mid \mathsf{let}\ x{=}e;; \\
v & :: & x \mid n \mid \mathsf{true} \mid \mathsf{false} \\
e & ::= & v \mid v_1\ op\ v_2 \mid \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
 & \mid & \mathsf{fun}\ x \to e \mid x_1\ x_2 \mid \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e_1\ \mathsf{in}\ e_2 \\
op & ::= & + \mid * \mid <
\end{array}$$

メタ変数 v は変数，整数値，そして真偽値を表すメタ変数である．式 e の文法が ML^4 のそれと少し変わっており，二項演算子の引数，条件式のガード部分，関数適用式の関数部分と引数部分には（式ではなく）変数か値しか取れないようになっている．これにより，先に挙げた式 (fun x -¿ (x + 1) * 2) (3 + 1) のような，引数部分に変数でも値でもない式 3+1 を取ることはできないようになっており，すべての部分式に名前をつけ，評価順序を明示することを強制している．

5.1.3　ML^4 から $\mathcal{C}$ への変換

変換 $\mathcal{I}$

コンパイラは ML^4 のプログラムを受け取り，そのプログラムを同等の振る舞いを持つ $\mathcal{C}$ のプログラムに変換する．この変換を行う関数 $\mathcal{I}$ を図 5.1 に示す．

関数 $\mathcal{I}$ は，式の構造に従って $\mathcal{I}$ を再帰的に適用し，かつ各部分式について他とカブらない変数（すなわち，fresh な変数）を生成してその部分式に束縛している．それぞれのケースの右辺が言語 $\mathcal{C}$ の構文に添っていることを各自確認されたい．例えば，$\mathcal{I}$ によって生成されたプログラムは，二項演算子 op の引数に必ず変数をとっている．

変換 $\mathcal{I}$ ではさらに式に現れるすべての束縛変数を fresh な変数名に置き換えることを行っている．これにより，異なる束縛変数が異なる名前を持つようになり，以降の変換の定義がシンプルになる．例えば，let x = 3 in let x = x + 1 in x は（それと等価な）let t1 = 3 in let t2 = t1 + 1 in t1 という式に変換される．[5]各束縛変数がどのような fresh な変数に付け替えられたかを記録しておくために，変換 $\mathcal{I}$ は写像 δ を持ち運ぶように定義してある．

5.1.4　簡単な最適化

コンパイラは生成されるターゲットプログラムの効率を改善するために**最適化** (*optimization*)[6]と呼ばれるプログラム変換を行う．言語 $\mathcal{C}$ の上で簡単な最適化をやってみよう．

[5] 生成された fresh な変数が順に t1，t2 であると仮定した．

[6]「最適化」という言葉は情報科学の分野では様々な意味を持つので注意が必要である．実数上の関数を与えられた制約の下で数値的な手法を用いて最小化または最大化する手法を研究する分野を**数値最適化** (*numerical optimization*)，離散値上の関数を与えられた制約の下で最小化または最大化する手法を研究する**組み合わせ最適化** (*combinatorial optimization*) 等があるが，これらをコンパイラで行われる最適化と混同しないこと．（もちろん，組み合わせ最適化や数値最適化を用いてコンパイラでの最適化を行うことはありうる．）

（以下の定義において，x, x_1, x_2, t_f, t_1 は他のどこでも用いられていない変数である.）

$\boxed{\text{Definition of } \mathcal{I}(e)}$

$$
\begin{array}{rcl}
\mathcal{I}_\delta(x) & = & \delta(x) \\
\mathcal{I}_\delta(n) & = & n \\
\mathcal{I}_\delta(\mathsf{true}) & = & \mathsf{true} \\
\mathcal{I}_\delta(\mathsf{false}) & = & \mathsf{false} \\
\mathcal{I}_\delta(e_1\ op\ e_2) & = & \ \ \mathsf{let}\ x_1 = \mathcal{I}_\delta(e_1)\ \mathsf{in} \\
 & & \ \ \mathsf{let}\ x_2 = \mathcal{I}_\delta(e_2)\ \mathsf{in} \\
 & & \quad x_1\ op\ x_2 \\
\mathcal{I}(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \ \ \mathsf{let}\ x = \mathcal{I}_\delta(e)\ \mathsf{in} \\
 & & \quad \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
\mathcal{I}_\delta(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \mathsf{let}\ t_1 = \mathcal{I}(e_1)\ \mathsf{in}\ \mathcal{I}_{\delta[x\mapsto t_1]}(e_2) \\
\mathcal{I}_\delta(\mathsf{fun}\ x \rightarrow e) & = & \mathsf{fun}\ t_1 \rightarrow \mathcal{I}_{\delta[x\mapsto t_1]}(e) \\
\mathcal{I}_\delta(e_1\ e_2) & = & \ \ \mathsf{let}\ x_1 = \mathcal{I}_\delta(e_1)\ \mathsf{in} \\
 & & \ \ \mathsf{let}\ x_2 = \mathcal{I}_\delta(e_2)\ \mathsf{in} \\
 & & \quad x_1\ x_2 \\
\mathcal{I}_\delta(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \rightarrow e_1\ \mathsf{in}\ e_2) & = & \mathsf{let\ rec}\ t_f = \mathsf{fun}\ t_1 \rightarrow \mathcal{I}_{\delta[f\mapsto t_f, x\mapsto t_1]}(e_1)\ \mathsf{in}\ \mathcal{I}_{\delta[f\mapsto t_f, x\mapsto t_1]}(e_2)
\end{array}
$$

$\boxed{\text{Definition of } \mathcal{I}(P)}$

$$
\begin{array}{rcl}
\mathcal{I}(e;;) & = & \mathcal{I}_\emptyset(e);; \\
\mathcal{I}(\mathsf{let}\ x{=}e;;) & = & \mathsf{let}\ t{=}\mathcal{I}_{\{x\mapsto t\}}(e);; \\
\mathcal{I}(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \rightarrow e;;) & = & \mathsf{let\ rec}\ t_f = \mathsf{fun}\ t_1 \rightarrow \mathcal{I}_{\{f\mapsto t_f, x\mapsto t_1\}}(e);;
\end{array}
$$

図 5.1: ML^4 から $\mathcal{C}$ への変換関数 $\mathcal{I}$.

無駄な束縛の除去

無駄な束縛の除去 (*elimination of redundant bindings*) は，その後使われることがなく，除去してもプログラムの意味を変えないとわかっている束縛を除去する変換である．例えば，プログラム let x = 3 in 4 は，x を 3 に束縛して 4 を返すが，束縛された x はその後一切使われないので，この束縛は**無駄** (*redundant*) である．これを束縛を行わない（多くの場合より効率のよい）プログラム 4 に変換するのが，無駄な束縛の除去である．

この変換は（というか，一般に多くの最適化は）何度か繰り返すことでより効率のよいプログラムを得ることが可能となる．例えば，プログラム let x = 3 in let y = x + 2 in 4 において，束縛変数 x は y の束縛先の計算を行う式 x+2 で使われているので無駄ではないが，束縛変数 y はその後使われていないので無駄である．そこで，無駄な y の束縛を除去すると，このプログラムは let x = 3 in 4 になるが，このプログラムにおいては x の束縛が無駄であるから，再度無駄な束縛を除去することにより，4 を得ることができる．

無駄な束縛の除去は図 5.3 に示す変換によって定義することができる．ここで $\mathbf{FV}(e)$ は e 中に現れる自由変数の集合である．この変換のキモは let $x = e_1$ in e_2 の e_2 中で束縛されている変数 x や let rec f = fun $x \to e_1$ in e_2 の e_2 中で束縛されている変数 f が無駄かどうかを判定する部分である．前者については x が e_2 に自由変数として現れておらず，かつ e_1 が関数適用の形をしていなければ，x の束縛を無駄であるとして除去する．前者の自由変数に関する条件は，x がその後使われないための十分条件になっている．後者の条件は e_1 が無限ループする式だった場合にプログラムの意味を変えないための条件である．例えば，let rec f x = f x in let x = f 3 in 4 という，必ず無限ループするプログラムを考えよう．もし後者の条件がなければ，x の束縛が（x が式 4 に自由に現れていないために）無駄だとして除去されてしまい，let rec f x = f x in 4 となって，無限ループしないプログラムになってしまう．最適化はあくまでプログラムの効率を上げるために行う変換なので，プログラムの意味を変えるのはマズい．これを防ぐために，x の束縛先が，無限ループに陥る可能性のある式である関数適用である場合は，束縛を除去しないようにしている．[7]

コピー伝播

コピー伝播 (*copy propagation*) は，let $x{=}y$ in e を $[y/x]e$ に置き換える変換である．ただし，$[y/x]e$ は e 中の x を y に置き換えた式を表す．これによって x の束縛が無くなるので，効率が良くなることが期待される．

[7] $\mathcal{C}$ では，ある式の評価が無限ループに陥るためには，関数適用を行わなければならない．(多分.)

Definition of $\mathcal{R}(e)$

$$
\begin{array}{rcl}
\mathcal{R}(x) & = & x \\
\mathcal{R}(n) & = & n \\
\mathcal{R}(\mathsf{true}) & = & \mathsf{true} \\
\mathcal{R}(\mathsf{false}) & = & \mathsf{false} \\
\mathcal{R}(x_1\ op\ x_2) & = & x_1\ op\ x_2 \\
\mathcal{R}(\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{if}\ x\ \mathsf{then}\ \mathcal{R}(e_1)\ \mathsf{else}\ \mathcal{R}(e_2) \\
\mathcal{R}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \begin{cases} \mathsf{let}\ x = \mathcal{R}(e_1)\ \mathsf{in}\ \mathcal{R}(e_2) & (\text{if } x \in \mathbf{FV}(e_2) \text{ or if } e_1 = x_1\ x_2) \\ \mathcal{R}(e_2) & (\text{otherwise}) \end{cases} \\
\mathcal{R}(\mathsf{fun}\ x \to e) & = & \mathsf{fun}\ x \to \mathcal{R}(e) \\
\mathcal{R}(x_1\ x_2) & = & x_1\ x_2 \\
\mathcal{R}(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e_1\ \mathsf{in}\ e_2) & = & \begin{cases} \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to \mathcal{R}(e_1)\ \mathsf{in}\ \mathcal{R}(e_2) & (\text{if } f \in \mathbf{FV}(e_2)) \\ \mathcal{R}(e_2) & (\text{otherwise}) \end{cases}
\end{array}
$$

Definition of $\mathcal{R}(P)$

$$
\begin{array}{rcl}
\mathcal{R}(e;;) & = & \mathcal{R}(e);; \\
\mathcal{R}(\mathsf{let}\ x{=}e;;) & = & \mathsf{let}\ x{=}\mathcal{R}(e) \\
\mathcal{R}(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e;;) & = & \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to \mathcal{R}(e);;
\end{array}
$$

図 5.2: $\mathcal{C}$ 上で無駄な束縛の除去を行う変換 $\mathcal{R}$.

$\boxed{\text{Definition of } \mathcal{R}(e)}$

$$
\begin{array}{rcl}
\mathcal{P}_\delta(x) & = & \begin{cases} \delta(x) & (\text{if } x \in \mathbf{dom}(\delta)) \\ x & (\text{otherwise}) \end{cases} \\
\mathcal{P}_\delta(n) & = & n \\
\mathcal{P}_\delta(\mathsf{true}) & = & \mathsf{true} \\
\mathcal{P}_\delta(\mathsf{false}) & = & \mathsf{false} \\
\mathcal{P}_\delta(x_1 \mathit{op}\, x_2) & = & \mathcal{P}_\delta(x_1)\, \mathit{op}\, \mathcal{P}_\delta(x_2) \\
\mathcal{P}_\delta(\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{if}\ \mathcal{P}_\delta(x)\ \mathsf{then}\ \mathcal{P}_\delta(e_1)\ \mathsf{else}\ \mathcal{P}_\delta(e_2) \\
\mathcal{P}_\delta(\mathsf{let}\ x = y\ \mathsf{in}\ e) & = & \mathcal{P}_{\delta[x \mapsto y]}(e) \\
\mathcal{P}_\delta(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \mathsf{let}\ x = \mathcal{P}_\delta(e_1)\ \mathsf{in}\ \mathcal{P}_\delta(e_2) \quad (\text{where } e_1 \text{ is not a variable}) \\
\mathcal{P}_\delta(\mathsf{fun}\ x \rightarrow e) & = & \mathsf{fun}\ x \rightarrow \mathcal{P}_\delta(e) \\
\mathcal{P}_\delta(x_1\ x_2) & = & \mathcal{P}_\delta(x_1)\ \mathcal{P}_\delta(x_2) \\
\mathcal{P}_\delta(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \rightarrow e_1\ \mathsf{in}\ e_2) & = & \mathsf{let\ rec}\ f = \mathsf{fun}\ x \rightarrow \mathcal{P}_\delta(e_1)\ \mathsf{in}\ \mathcal{P}_\delta(e_2)
\end{array}
$$

$\boxed{\text{Definition of } \mathcal{P}(P)}$

$$
\begin{array}{rcl}
\mathcal{P}(e;;) & = & \mathcal{P}_\delta(e);; \\
\mathcal{P}(\mathsf{let}\ x{=}e;;) & = & \mathsf{let}\ x{=}\mathcal{P}_\delta(e) \\
\mathcal{P}(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \rightarrow e;;) & = & \mathsf{let\ rec}\ f = \mathsf{fun}\ x \rightarrow \mathcal{P}_\delta(e);;
\end{array}
$$

図 5.3: $\mathcal{C}$ 上でコピー伝播を行う変換 $\mathcal{P}$.

定数畳み込み

関数のインライン化

5.1.5 クロージャ変換

5.1.6 仮想コードの生成

5.1.7 アセンブリ生成

5.2 扱っていないトピック

5.2.1 フロー解析

5.2.2 真面目なレジスタ割り当て

5.2.3 命令選択

5.2.4 より先進的なコンパイラの話

第6章　字句解析と構文解析

今から書きます．ごめんね．

第7章　おわりに：俺たちのプログラミング言語処理系はまだ始まったばかりだ！

まだ書いてません．ごめんね．

関連図書