{% include head.html %}

各モジュールの機能 (1)

実装するインタプリタは 6 つのモジュール注から構成される．本節ではそれぞれのモジュールについて簡単に説明する．

モジュールについて: 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，- 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）- 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．

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が定義されている．型定義が含まれている．以下に `syntax.ml` の中身を示す．

{% highlight ocaml %} {% include_relative interpreter/src/syntax.ml %} {% endhighlight %}

以下では `Syntax` モジュールで定義されている型は変数を説明する．実際に `syntax.ml` と前出の BNFを見ながら読んでみてほしい．- `id` は変数の識別子を示すための型で，その実体は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やコンパイラでは，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` 型に関しては前出の BNFでのシンタックスの定義を（括弧式を除いて）そのまま写した形の宣言になっている．例えば，式 `3+x'` は `exp` 型の値 `BinOP(Plus, ILit 3, Var "x'")` で表現される．- `typ` 型は型推論器を実装するときに用いる「型を表す型」である．今のところは無視して良い．

式の抽象構文木を表す値（すなわち，`exp` 型の値）は，プログラムを表す文字列から，字句解析と 構文解析と呼ばれる処理によって生成される．これらの処理については後述するが，- `Lexer` モジュールには字句解析のための型や関数が定義されており，- `Parser` モジュールには構文解析のための型や関数が定義されており，- `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．

Environment モジュールと Eval モジュール：環境と解釈部

式の表す値

Eval モジュールはインタプリタの動作のメイン部分であり，字句解析と構文解析によって生成された構文木を解釈する．（したがって，この部分をインタプリタの 解釈部と呼ぶ.）解釈部に動作によって，言語処理系は定義される言語のセマンティクスを定めている．以下に Eval モジュールの中身を示す．

{% highlight ocaml %} {% include_relative interpreter/src/eval.ml %} {% endhighlight %}

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の一つは，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，ヴァリアントなどが値として扱える．

言語によっては，このとき 式の値 *(expressed value)* の集合と 変数が指示する値 *(denoted value)* を区別する必要がある．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，実際に今回のインタプリタの範囲では，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．例えば，C 言語では，変数は，値そのものに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．

MiniML1 の場合, 式の値 expressed value の集合は $\{..., -2, -1, 0, 1, 2, 3, ...\} \oplus$真偽値の集合 であり, denoted value の集合は expressed value の集合に等しい．ここで，$\oplus$ は直和を示している．

eval.ml には値を表す OCaml の型である exval と dnval が定義されている．インタプリタ内では，MiniML の値をこれらの形の（OCaml の）値として表すことになる．型宣言は，初めは以下の通りになっている．

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

exval が expressed value の型，dnval が denoted value の型である．

環境

解釈部を構成する上では，式を評価する際に，各変数の値が何であるかを管理することが重要である．そのために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 環境 *(environment)* と呼ばれるデータ構造を受け取り，抽象構文木が表す式の評価結果を計算する 環境渡しインタプリタ *(environment passing interpreter)* という方法である．

言語処理系の文脈における 環境 *(environment)* とは，変数から denoted value への関数（もしくはこの関数

を表現するデータ構造）のことである．環境は変数の denoted value への 束縛 *(binding)* 1を表現する．束縛とは各変数を何らかのデータ（ここでは denoted value）に結びつけることである．プログラムにおいて，各変数が何に束縛されているか（= 各変数の値が何であるか）を，環境で表現するのである．例えば，$\{x \mapsto 1, y \mapsto 3\}$ という写像は環境である．この環境は変数 x を 1 に，y を 3 に写像しており，変数 `x` の値が `1` であり，変数 `y` の値が `3` であることを表している．

1: 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．値 v が変数 x に束縛されている とはいわないので注意すること．

OCaml で MiniML の言語処理系を実装する上では，環境をどのような型の値として表現するかが重要である．環境を実装する上では，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，これらの環境から送出されか可能性のある例外は，`environment.mli` に以下のように定められている．{% highlight ocaml %} {% include_relative interpreter/src/environment.mli %} {% endhighlight %} - 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以下の `environment.ml` である．{% highlight ocaml %} {% include_relative interpreter/src/environment.ml %} {% endhighlight %} 環境のデータ表現は，変数と，その変数が束縛されているデータのペアのリストである．例えば上に出てきた環境 $\{x \mapsto 1, y \mapsto 3\}$ はリスト `[(x,1); (y,3)]` で表現される．ただし `environment.mli` では型 `'a t` が定義のない抽象的な型として宣言されているので，環境を使う側からは環境の実体がこのように実装されていることを使うことはできず，環境の操作は `Environment` モジュール中の関数を介して行う必要がある．（例えば，環境 `env` に対して `match env with [] -> ... | hd::tl -> ...` のようにリストのパターンマッチを適用することはできない．）

環境の作り方の例を見てみよう．以下は後述する `cui.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．{% highlight ocaml %} let initial_env = Environment.extend “i” (IntV 1) (Environment.extend “v” (IntV 5) (Environment.extend “x” (IntV 10) Environment.empty)) {% endhighlight %} `empty` と `extend` を用いて `i`，`v`，`x` が，それぞれ `1`，`5`，`10` に束縛されている環境を作成している．`cui.ml` は `Environment` モジュールの外側にいるので，`empty` と `extend` を用いる際には `Environment.empty`, `Environment.extend` のように用いている．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．

.ml ファイルと.mli ファイルの関係について（あるいは,「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 モジュールがどう動作するかを決定する 実装 *(implementation)* を定義し，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する インターフェイス *(interface)* を宣言する．2 中身を見てみると，environment.ml には Environment モジュールがどう動作するかが記述されており，environment.mli は，このモジュールの使われ方が型によって表現されている．

2: 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．

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 型として定義し，'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探索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において，'a t 型がリストであることに依存した記述を行うことが可能となる．そのようなプログラムを書いてしまうと，二分木の実装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．

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．eval.ml では，前者を apply_prim, 後者を eval_exp という関数として定義している．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_decl は MiniML1 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．

Cui モジュール

メインプログラム main.ml は Cui モジュール中で定義されている read_eval_print という関数を呼び出している．Cui モジュールは以下のようになっている．{% highlight ocaml %} {% include_relative interpreter/src/cui.ml %} {% endhighlight %} 関数 read_eval_print は，+ 入力文字列の読み込み・構文解析 + 解釈 + 結果の出力という処理を繰返している．

（以下の説明は，わからなければとりあえず飛ばしてもよい.）まず，let decl = の右辺にある式

```
Parser.toplevel Lexer.main (Lexing.from_channel stdin)
```

を見てみよう．この式は，標準入力から入力された文字列を抽象構文木（すなわち Syntax.exp 型の値）に変換して返す．

- 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 型の値を取り，バッファに貯められた文字列に字句解析と構文解析を適用して抽象構文木を返す．
- Lexer.main と Parser.toplevel は，それぞれ ocamllex と Menhir によって自動生成された関数である．前者は字句解析を行うメインの関数，後者は構文解析を行うメインの関数である．詳しくは次節を参照のこと．
- 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の入力を貯め込むバッファを作るため，Lexing.from_channel を使ってバッファを作っている．
- pp_val は eval.ml で定義されている，値をディスプレイに出力するための関数である．

標準ライブラリ

OCaml の標準ライブラリのドキュメントを読むと，標準ライブラリの使い方が分かる．+ また，OCaml の処理系であらかじめ使える関数群はStdlibに定義されている．これは目を通しておくとよいだろう．+ List, Map, Set, Hashtblは，リスト操作，写像操作，集合操作，ハッシュマップ操作のためのライブラリである．これらは読んでおくとよい．+ なにかメッセージを出力したいときにはPrintf, Formatが役に立つ．Format ははじめはわけがわからないかもしれないが，使い慣れると良い．+ Arg（コマンドライン引数の parse），Array（配列操作）, Filename（ファイル名に対する操作）, Lazy（計算の一部を遅延させる操作）, Random

（疑似乱数）, String（文字列関係）, Sys（システムとのインタフェース）, Gc（GC やメモリ管理の操作）とかは使いどころが結構あるかもしれない.

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.