



ML 演習 第 5 回

おおいわ
May 8, 2001



お願い

- メールアドレスを確認してください
 - ml-query の返事が不達で戻ってきます
 - 特に、foobar@cses0g.is 形式が多いです

今回の内容

- モジュールと分割コンパイル
- 言語処理系の実装 (1)
 - 形無し関数型言語のインタプリタの作成
 - 文法と構文木
 - Call-by-value 戦略に基づく式の評価



ocamlc (1)

- Ocaml のコンパイラ
 - モジュール単位の分割コンパイルサポート
 - unix の実行形式ファイルを作成
 - 複数の backend
 - ocamlc: バイトコードコンパイラ
 - ocamlc.opt: ネイティブコードコンパイラ

ocamlc (2)

■ 拡張子一覧

■ ソースファイル

- .ml → module の実装 (structure)
- .mli → module のインタフェース (signature)

■ オブジェクトファイル

- .cmo → 実装のバイトコード
- .cmi → インタフェース定義のバイトコード
- .cmx → 実装のネイティブコード

分割コンパイル (1)

■ .ml と .mli

- 実装とインタフェースをそれぞれ記述
 - `module SomeThing :`
 `sig [someThing.mli の内容] end`
 `= struct [someThing.ml の内容] end`
 に相当
- .mli をコンパイル → .cmi を生成
- .ml をコンパイル → .cmi が無ければ
制約無しで生成、あれば型チェック

分割コンパイル (2)

■ 例

- mySet.mli, mySet.ml
 - module MySet の定義 (内容はほぼ第4回の実装)
- uniq.ml
 - メインプログラムのモジュール

分割コンパイル (3)

■ 実行例 (1)

```
% ocamlc -c mySet.mli
```

```
% ocamlc -c mySet.ml
```

```
% ocamlc -c uniq.ml
```

```
% ls -F *.cm*
```

```
mySet.cmi mySet.cmo uniq.cmi uniq.cmo
```

```
% ocamlc -o myuniq mySet.cmo uniq.cmo
```

```
% ls -F myuniq
```

```
myuniq*
```

分割コンパイル (4)

■ 実行例 (1)

% ./myuniq

OCaml

Standard ML

C++

OCaml

^D

C++

OCaml

Standard ML

%

分割コンパイル (5)

■ .cmo のインタプリタでの利用

```
# #load "mySet.cmo";;
```

```
# MySet.empty;;
```

```
- : 'a MySet.set = <abstr>
```

```
# MySet.remove_top;;
```

```
Unbound value MySet.remove_top
```

言語処理系の作成

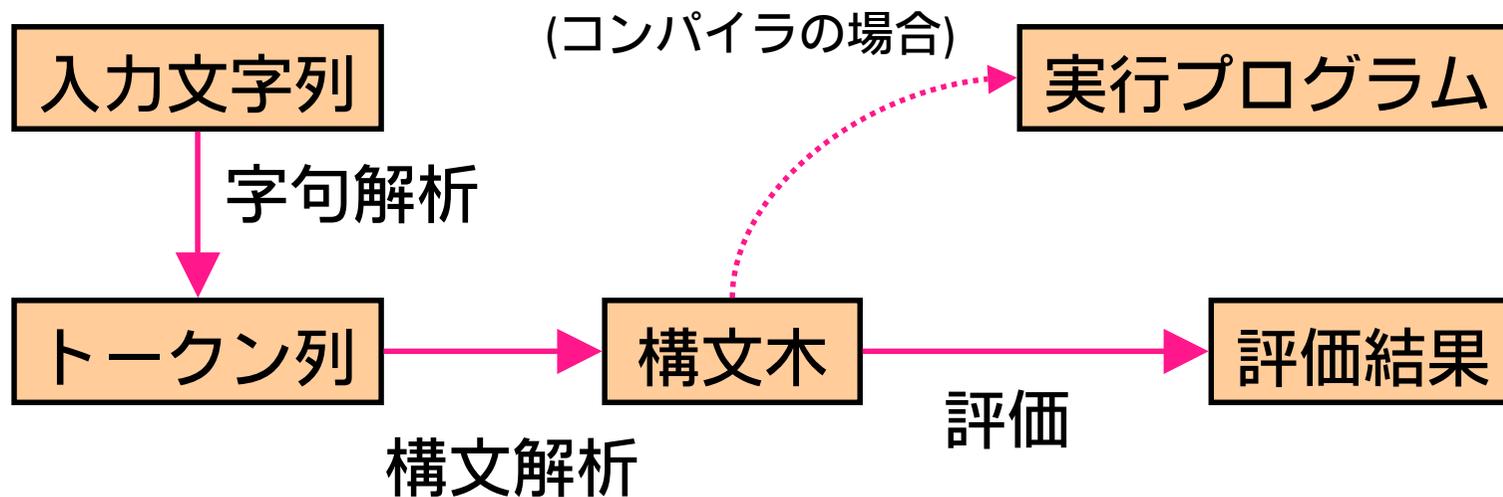
- 今後3回の予定
 - 第5回: 基本的なインタプリタの作成
 - 形無しMLの処理系の作成
 - 第6回: インタプリタの様々な拡張
 - 式の評価順序に関する考察
 - 第7回: 言語処理系と型システム
 - ML 風の型推論の実装

今回の言語の構文

- OCaml の小さな subset
 - データ型: int, bool, 関数, pair, list
 - 構文: 定数, 加減乗除, =, pair, ::, if, fun, 関数適用, match, let, let rec
- とりあえず今回は動的な型チェックで実装 (Scheme 風に)

言語処理系の構造

- 入力: プログラム文字列
- 出力: 評価結果



字句解析

- 入力文字列を「単語」に切り出す

- 例: (fun x -> x * 5) 3

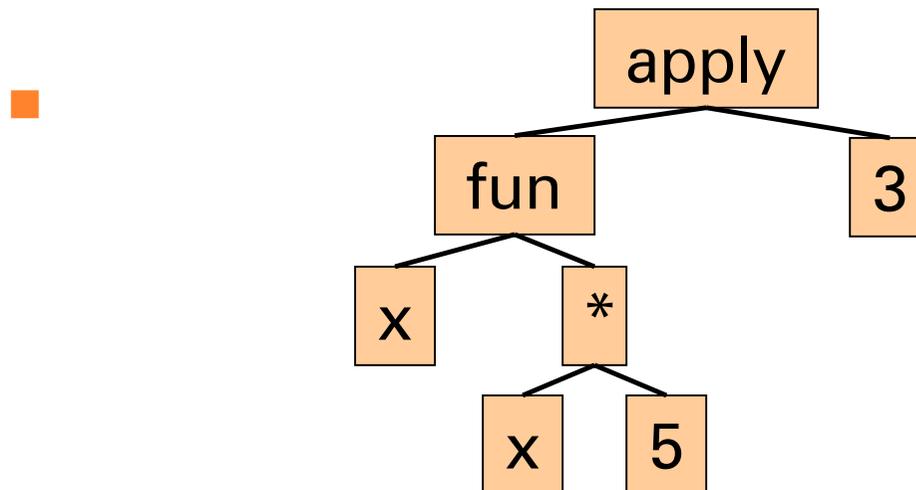
- 出力: (fun x -> x * 5) 3

- ツール: lex, flex, ocamllex etc...

構文解析 (1)

- 字句の列から文法解釈して構文木に

- 例: (fun x -> x * 5) 3



構文解析 (2)

- ツール: yacc, bison, ocamlyacc, etc...
- 今回は字句・構文解析はこちらで提供します
 - モジュール MiniMLReader

Mini-ML の値

- ML の subset (type mlvalue: miniML.ml)
 - 整数 (0, 1, 2, ...) `Int x`
 - 論理値 (true, false) `Bool b`
 - リスト `Nil, Cons(x, xs)`
 - ペア `Pair(x1, x2)`
 - 関数 `Closure([pattern, exp], env)`

仮引数

本体の式

環境

Mini-ML の式 (1)

- Caml 上での表現: type expr
 - 定数式 `Const(x : mlvalue)`
 - 変数参照 `Var(x : string)`
 - 整数演算 `Plus(e1, e2)`
`Minus(...), Times(...), Div(...)`
 - 等値比較 `Equal(e1, e2)`
 - リストの生成 `ConsExp(e1, e2)`
 - ペアの生成 `PairExp(e1, e2)`

Mini-ML の式 (2)

- if 文 $\text{IfExp } (e1, e2, e3)$
- lambda 抽象 $\text{LambdaExp } [\text{IdentPtn } id, e]$
- 関数適用 $\text{App}(e1, e2)$
- match $\text{MatchExp}(e, \text{match_list})$
 - match_list: $[pattern1, exp1; pattern2, exp2; \dots]$
- let 束縛 $\text{LetExp}([\text{IdentPtn } id, e1], e2)$
 $\text{LetRecExp } ([\text{IdentPtn } id, e1], e2)$

optional 課題のための拡張。
とりあえず気にしなくてよい。

Mini-ML のパターン言語

- 定数パターン `ConstPtn(x)`
- 変数束縛パターン `IdentPtn(ident)`
- 任意パターン `IdentPtn("_")`
- リストパターン `ConsPtn(ptn1, ptn2)`
- ペアパターン `PairPtn(ptn1, ptn2)`

環境

- 自由変数と値の間の束縛関係を記憶
 - 評価の進行に応じて拡張される

例: let x = 5 in let y = 3 in x + y

- この下線部を評価している時点での環境は
 $\{ x \rightarrow 5, y \rightarrow 3 \}$
- Mini-ML で環境を拡張する構文の例:
 - App, MatchExp, LetExp, LetRecExp, ...

環境の表現

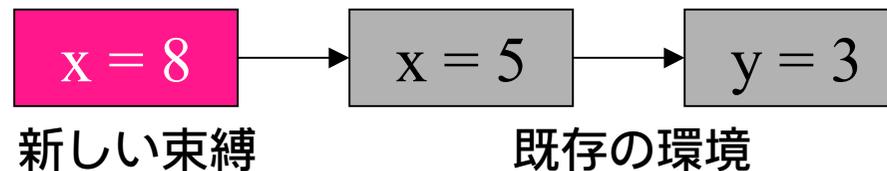
- 束縛 (string * mlvalue ref) のリスト
 - Scheme と違い、一回束縛した値は書き換わることが無いので、原理的には mlvalue でいいのだが、LetRecExp の実装の都合上 mlvalue ref の方が都合がいいのでこうしてあります。

環境の実装

- `get (miniMLInterp.ml)`

- `eval (LetExp の節)`

```
let rec eval env LetExp([IdentPtn id, e1], e2) =  
  let v1 = eval env e1 in  
  eval ( (id, ref v1) :: env ) e2
```



式の評価 (1)

- **λ 抽象**
 - その時点での環境を保存
- **関数適用**
 - 実引数を現在の環境で評価
 - 仮引数を実引数の評価結果に束縛し、
それで保存されていた環境を拡張
 - 関数本体を拡張した環境で評価

式の評価 (2)

■ $f := \text{let } x = 5 \text{ in } (\text{fun } y \rightarrow x + y)$

■ f :

$y \rightarrow x + y$	$x = 5$
-----------------------	---------

■ $\text{let } x = 3 \text{ in } f\ x$

■ この時点での環境 :

$x = 3$

■ 実引数 "x" の評価結果 3 を y に束縛

■ 環境

$y = 3$

 \rightarrow

$x = 5$

 で " $x + y$ " を評価

式の評価 (3)

- Let 式:

- まず値を計算
- 変数に束縛して環境を拡張
- in 節の式を拡張された環境で評価

式の評価 (4)

■ LetRec 式:

- 先に環境を拡張 (中身の値は参照禁止)
- 拡張された環境で式を評価
- その値を束縛
- in 節の式を拡張された環境で評価

→ 自分自身が束縛された環境を参照

パターンマッチ (1)

- パターンと値を見比べて束縛を作る
- データ構造パターン (ConsPtn) とのマッチは内部を再帰的に調査

- 例:

ConsPtn (IdentPtn x, ConstPtn (Nil)) と

Cons (Int 1, Nil)

→ 結果は { x = 1 }

パターンマッチ (2)

ConsPtn (IdentPtn x, ConstPtn (Nil))
Cons (Int 1, Nil)

1. トップのデータ構造の比較:
ConsPtn \leftrightarrow Cons : 内部が合えば合致
2. 第1要素の比較:
IdentPtn x \leftrightarrow Int 1 : x を 1 に束縛
3. 第2要素の比較:
ConstPtn (Nil) \leftrightarrow Nil : 合致

構文解析モジュール (1)

- Mini-ML 用パーサの使い方:
 - .cmo ファイルを3つ読み込む
 - miniMLReader.ml のコメント参照
 - ファイルは演習の URL を参照
 - なお、miniML.ml の定義を変更した場合、Makefile を用いて再コンパイルの必要がある場合があります。

構文解析モジュール (2)

■ 例

```
# let exp = mlexp_of_string "fun x -> x + 1";;
```

```
- : MiniML.expr =  
  LambdaExp
```

```
  [IdentPat "x", Plus (Var "x", Const (Int 1))]
```

```
# eval [] (mlexp_of_string "5 + 3");;
```

```
- : MiniML.mlvalue = Int 8
```

- $\text{fun } x \ y \rightarrow x + y$ や $\text{let } f \ x = x + 3$ などは `LambdaExp` などの組み合わせに展開されます。

課題1

- miniMLInterp.ml のインタプリタに、関数適用 (App) と LetRec 式 (LetRecExp) に対する実装を追加せよ。
 - それぞれ failwith “...” になっている所を各自の実装で埋めてください。
 - 実装方針はここまでの説明を参照。

課題2

- パターンと値をとって、pattern match 時に生じる束縛を生成する関数
match_pattern :
pattern → mlvalue →
(string * mlvalue ref) list
を作成し、
eval に MatchExp に対する実装を追加
せよ。

課題3 (optional)

- LetExp, LetRecExp の実装をパターンと and 節に対応させよ。
 - 束縛のタイミングに要注意。
 - [IdentPtn id, e1] と書いてあったところに [pattern1, exp1; pattern2, exp2] という形で複数パターンが与えられます。
- Lambda 式を複数パターン選択 (function 式) に対応させよ。
 - もちろん実際は App の書き換えの方が重要。



提出方法

- 〆切: 2001年5月21日 (月) 24:00
- 提出先: ml-report@yl.is.s.u-tokyo.ac.jp
- 題名: Report 5 学生証番号