Functional and logic programming lab 5th report

Yoshiki Fujiwara, 05-191023

解いた順番で考察を書いたため、順番が前後しています。すいません。Q1-Q7 まで全て解いていて、Q5 だけ一番最後に記述しています。

Q1-Q4 は 4 というディレクトリにまとめています。

Q5 は 5 というディレクトリに実装しています。このディレクトリで Q6,Q7 の機能も実装できています。Q6 は 6 というディレクトリに実装しています。Q7 は 7 というディレクトリに実装しています。

また、Q5 と Q7 では Q6 までで作った部分を壊したりしたので、別で提出しています。

1 Q.1: example.ml の実行

1.1 動作例と考察

1. ocamlyacc parser.mly

この操作によって、parser.ml と parser.mli が作られる。

ocamlyacc では、トークン列から抽象構文木にしている。文法を木にしている。ここでは、parser.mly を受け取って、parser.ml と parser.mli を作っている。

parser.mly の中の $\mathrm{arith}_e expr$ を $expr$ にすると、当然だが、文法の規則の優先順位がわからなくなり、$reduce/reduceconflict$ が起こる。

ocamlyacc の出力のもう一つ、parser.output には、オートマトンの各状態に対するものが記載されている。次に何がくると reduce をして、何がくると shift をするかなどが記載されている。

2. ocamllex lexer.mll

この操作で、lexer.ml が作られる。これはプログラムから字句解析器が作られている。lexer.mll には、プログラムを単なる文字列とみて、その文字一つ一つに対して、どうしていくのかが記載されている。

3. ocamlc –c syntax.ml

syntax.ml から実行ファイル (.cmo と.cmi) が作られる。syntax.ml には、関数の型が記載されている。

4.ocamlc –c parser.mli ocmalc –c parser.ml

parser に対する実行ファイルの作成。

5. ocamlc -c lexer.ml

3-5 をこの順番で行う理由は依存関係による。

プログラムが、字句解析器（5 でコンパイルしたもの）を通って、トークン列となり、

トークン列が、構文解析器（4 でコンパイルしたもの）を通って、抽象構文木となり、

抽象構文木にしたがって、計算するプログラム（3 でコンパイルしたもの）を介して、評価が行われるからである。

6.ocamlc –c example.ml

実際に処理を行うプログラムの作成。これは標準出力と標準入力を先ほどコンパイルしたプログラムたちと繋げている。

7. ocamlc –o example syntax.cmo parser.cmo lexer.cmo example.cmo

ここでも依存関係を気にしなければならない。

2 Q2: 引き算、乗算、除算の定義

2.1 動作例

Listing 1 動作例

```
# 3-2;;
- = 1
# 2 * 3;;
- = 6
# 6/2;;
- = 3
```

2.2 考察

変えるべきところは、eval.ml だけであった。

lexer.mll の中で、-, *, / についてのプログラムの認識はすでに作られており、parser の中でそれに対する優先順位と構文木はすでに書かれており、syntax.ml でそれに対する型も書かれていた。

本来ならば、上記全てを変える必要がある。

eval.ml では、expr 型と expr 型をもらって評価結果を返せば良い。

expr 型を評価するときの環境は現在の環境である。

3 Q3: let 文の定義

3.1 動作例

Listing 2 動作例

```
# let x = 3 in
  x + 3;;
  - = 6
# let x = 4;;
  x = 4
# x + 2;;
  - = 6
#let x = 2 in
 let y = x + 2 in
 y + 3;;
 - = 7
```

3.2 考察

今回の実装についても、eval.ml の部分を実装するだけで十分であった。なぜなら、その他の部分については、すでに配布資料で実装が行われていたからである。

let in 文については ELet で実装し、let 文については、CDecl で実装した。ELet の処理については、eval_expr の中で定義されていて、CDecl の処理については、eval_command の中で定義されている。

eval_expr では、評価結果の値 (value 型) のみを返し、eval_command では、「変数名、新しい環境、評価結果」を返す。このうち、変数名と、評価結果は標準出力に出力される。

よって、let in 文は新しい環境を返す必要がない上に、let in 文の中の変数は、標準出力に出してはいけないので、eval_expr で定義するべきである。

let 文は逆に、eval_command で定義するべきである。

letin 文についての実装は以下のようになっていて、

$$eval\_expr(extend\ e1\ (eval\_expr\ env\ e2)\ env)\ e3$$

let x = 3 in x+3;; では、e1 に x、e2 に 3、e3 に x+3 が対応している。

$(eval\_expr\ \ env\ \ e2)$ で、e2 の expression を評価して、value 型にする。ここで、昔の環境を使っていることに注意する。この値を環境に加えたものを $(extend\ e1\ (eval\_expr\ env\ e2)\ env)$ で作り、そのもとで、e3 を評価している。

let 分についての実装は以下のようになっていて、

$$CDecl(e1, e2) -> (e1,\ (extend\ e1\ (eval\_expr\ env\ e2) env),\ eval\_expr\ env\ e2)$$

let x = 4;; では x が e1 に、4 が e2 に対応している。

eval_command で返す第一要素には、変数名を返せば良いので、e1 を返している。

eval_command で返す第二要素には、新しい環境を返せば良いので、$(extend\ e1\ (eval\_expr\ env\ e2) env)$ を返している。ここで、変数の評価は、元の環境を使って行なっていることに注意する。

第三要素は、評価結果を返せば良い。

4 Q4: Bool の計算

4.1 動作例

Listing 3 動作例

```
# true && true;;
  - = true
# true && false;;
  - = false
# false || true;;
  - = true
# false || false;;
  - = false
# false || if 1 = 1 then false else true;;
- = false
# true && let x = true in x;;
- = true
# false || 1 = 1;;
```

```
- = true
```

4.2 考察

まず、lexer.mll で、 と || に対する token を定義して、parser.mly で文法を定める。そして、syntax.ml で型を定め、eval.ml でどのように評価するのかを定める。

bool の演算については、評価の順番に気をつけなくてはならない。動作例の最後の三つから分かるように、評価は let 文や if 文よりも後に行われる。すなわち、構文木においては、上の方で早めに分岐させなければならない。

Eval.ml の Eq や Lt も bool が定義できるように変えている。

5 Q6: let 文を同時に定義できるような実装

5.1 動作例

Listing 4 動作例

```
# let x = 10
  let y = x+1
  let z = x*y;;
  x = 10
  y = 11
  z = 110
# let x = true
  let y = false
  let z = x && y;;
  x = true
  y = false
  z = false
```

5.2 考察

let 文を連続して入力できるようにした。

新たに letexpression というのを定義して、再帰的に evaluation を行えるようにした。

この時、型に関して、letexpression は command 型であることに注意する。

eval.ml のなかの、LLet の評価については、環境を更新しつつ新しい変数を評価しなければならないので、let in 文を用いて書いている。この時、eval_command 関数を再帰的に定義させる必要がある。

出力するまでに値を覚えておく必要があったので、command 型の第一要素と第三要素に変更を加えた。具体的にはリストとして働くようにした。それに応じて、標準出力と対応させている main.ml の部分も、リストから出力できるように変更を加えた。

6 Q7: let and 文を定義できるような実装

6.1 動作例

Listing 5 動作例

```
# let x = 2 and y = 4 and z = 3;;
x = 2
y = 4
z = 3
# x;;
- = 2

# let x = 10;;
x = 10
# let x = 50
and y = x * 2;;
x = 50
y = 20

# let x = 10;;
x = 10
# let x = 50
and y = x * 2
in x + y;;
- = 70
```

letand については Q6 の let 文の環境が保存されないものであるので、Q6 の実装を少し変えることと、let and のための letandexpression という非終端記号を作れば良い。通常の let 文との区別を行うために、toplevel で and が入るか否かの判定を行なっている。letand のために作成した関数は、AndLet のみである。これは再帰的に呼び出されるものである。

また、in の扱いについては、再考しなければならず、構文木の変更を行った。in の前は

評価されないことに着目し、inbefore という in の前にくる let 文のための非終端記号を作成した。InEval という関数は、inbefore で定義された型を元に関数の評価を行うものである。

この InEval という実装のため、eval_expr に環境を返して欲しかったため、type value の中に (name * value) list を作成した。

それを踏まえて、let and in 文のために、TempList と TempListAdd を作成した。

TempList は in の前の文で作られる環境を一時的に保存しておくリストを作るもので、TempListAdd はそれに新たに加えていく関数である。

InEval は、eval_expr で受け取った環境に、現在の環境を append した環境で、評価を行うという操作を行えば良い。

7 Q5: エラー処理

7.1 動作例

Listing 6 動作例

```
# a;;
Error: Unbound value a
# 1 + true;;
Error: This expression has type bool but an expression was expected of type
         int
# let x = ?;;
Unknown Token: ?
# let x = 2 and let z = 3;;
parse error near characters 14-17
#
```

7.2 考察

まず、例外が発生しても interpreter が終了しないようにするために、main.ml を改良する。

interpreter が終了しないようにするためには、try 文を使えばよく、try 文で「構文解析器、字句解析器、評価器」の部分を囲えば良い。

評価器のエラーについては、raise EvalErr としているので、try 文の with で EvalErr に対応するものを作れば良い。

字句解析器と構文解析器のエラーについては、failwith で処理を行っているため、try

文の with 以下で Failure に対応するものを作れば良い。

以上のことをすると、例外が発生しても interpreter が終了しないようにすることができる。

次に、Error 出力について、評価器のエラーは、match 文のそれぞれについてエラー出力を行えば良い。Ocaml のエラーコードをそれぞれ見ることで実装を行った。

構文解析器のエラーはデフォルトで実装されていたものを使用し、字句解析器のエラーについては、何文字目でおかしいのかを表示できるようにした。