レポート2(インタプリタ)

1029289895 尾崎翔太

2018/7/5

1　はじめに

ソースコード単位で解説していくのではなく、課題単位で解説していく。また、説明はできるだけその時点における説明にしようと思うので、ソースコードとの整合性が取れない場合がある。例えば、ソースコードにおいてLTExprはCmpExprになっているが、説明ではLTExprが登場する。最後に、課題には存在しなかったが、デバッグのしやすさから減算と、比較のイコールを実装している（ただし、減算は-と第二オペランドの間に空白を入れないときちんと動作しない)。

2　各課題の説明

2.1　Exercise 3.2.1

テストプログラムとその実行は以下である。

```
# if (3 * 4) < 15 then 1 + 2 * 3 else 0;;
val - = 7
```

これによりif文、加算、乗算、比較、その優先順位が正しく動作していることがわかる。また、問題文にあるようにiiを2、iiiを3、ivを4に束縛した環境を大域環境にすると

```
# iv + iii * ii;;
val - = 10
```

となる。

2.2　Exercise 3.2.2

2.2.1　設計方針

read-eval-printループ内でエラーを検出して、エラーの種類に応じてエラーメッセージを表示したあとにread-eval-printループに戻る。

2.2.2 実装の詳細

main.ml の 7〜10 行目に print_error_and_go 関数を定義している。これは string を受け取って、それを表示したあとに read_eval_print 関数を呼び出す。これは、エラー処理をまとめたものである。そして main.ml の 12 行目に try があって、47 行目に with があって、エラーをパターンマッチして、表示するメッセージを print_error_and_go 関数にわたす。エラーの種類は eval で発生するもの、parser で発生するもの、lexer で発生するもの、正体不明の四つに分類した。

2.3 Exercise 3.2.3

2.3.1 設計方針

単純に & & と || を認識できるようにして、それらに対応する文法規則を追加する。評価においては、第一オペランドを評価した時点で値が確定するなら第二オペランドは評価しない。そのため、通常の二項演算子とは異なる関数が必要となる。

2.3.2 実装の詳細

lexer.mll の 38 行目は「& & 」を AND というトークンに、39 行目は「||」を OR をいうトークンに変換する規則である。そして、優先順位を考慮して、LTExpr の上に AndExpr、AndExpr の上に OrExpr というようになっている。評価に関しては、他の二項演算子と違って、オペランドを評価したりしなかったりするので、区別するために BinLogicOp という新たな型を作った。この型のコンストラクタは And と Or の二つである。そして、eval.ml の 64〜77 行目に apply_logic_prim 関数を定義している。これは、BinLogicOp と exval と exp を引数にとって、exval を返す。op が And で arg1 が BoolV false なら exp2 は評価せずに BoolV false を返す。op が And で arg1 が true なら exp2 の評価結果を返す。op が Or の場合も同様である (true と false は逆だが)。なお、オペランドが bool の値をもってなければエラーを吐く。また、eval_exp 関数に BinLogicOp の場合を追加して、第一オペランドを評価してから apply_logic_prim 関数を呼び出している。最後に、apply_logic_prim 関数と eval_exp 関数は相互再帰の形になったので、and でつないでいる。

2.4 Exercise 3.2.4

2.4.1 設計方針

lexer.mll に新たなルールを追加して、再帰的に呼び出せるようにする。ただし、コメントのネストがきちんとできるようにコメントの深さを管理する。

2.4.2 実装の詳細

lexer.mll の 57 行目以降に comment というルールを追加した。このルールは整数を一つ受け取る。この整数はコメントの深さを表していて、この「(∗ 」が現れるたびに 1 増え、「∗) 」が現れるたびに 1 減る。ルール main で「(∗ 」が現れたときはルール comment を 0 で呼び出す。そして、0 のときに「∗)」が現れたときはルール main を呼び出す。

2.5 Exercise 3.3.1

教科書通りにしただけなので、設計方針等はない。テストプログラムは以下である。

```
# let x = 1 + 2;;
val x = 3
# let y = 1;;
val y = 1
# let y = x + 1 in y * 5;;
val - = 20
# y;;
val - = 1
```

let 宣言によって x が 3 に束縛されていることがわかり、let 式もきちんと動作していることがわかる。さらに、let 式で束縛した y はスコープの外では見えないことがわかる。

2.6 Exercise 3.3.2/Exercise 3.3.4

2.6.1 設計方針

let 宣言の列は and でつながれた let 宣言の列の列である、と考える。よって、let 宣言の全体像としては、リストのリストのようになる。また、and の方は評価に使う環境と束縛を付加する環境が違うことに注意する。let 式の方は、let 宣言と大差ないので、let 宣言を実装してからそれをいじって実装することにする。

2.6.2 実装の詳細

主に let 宣言について説明する。syntax.ml の 45 行目のように、let 宣言を表すコンストラクタは Decls of ((id * exp) list) list である。設計方針で述べたように、外側のリストが let 宣言の列、内側のリストが and でつながれた let 宣言を表している。lexer については「and」を LETAND というトークンに変換する規則を追加した。parser は parser.mly の 28〜36 行目に LetDecl と LetAndDecl という非終端記号を追加している。中身はどちらも同じような格好をしていて、打ち切るか続くかのどちらかとなっている。評価については eval.ml の 214〜234 行目で行っている。複雑だが、要するに ((id * env * value) list) list を返している。外側のループは単に外側のリストを構成しているだけで、いろいろしているのは内側のループである。内側のループは、and でつながれた let 宣言において同じ変数に束縛していないかを、変数名のリストを持ちまわることでチェックしている。また、式の評価は外側のループにおける環境で行っていて、その束縛を付加する環境は内側のループのものである。そして、内側のループを抜けるときにその環境を外側のループの環境に代入する。要するに、and でつながれた let 宣言によって起こった束縛はまとめて環境に付加するということである。main.ml においては 15〜46 行目が関係している部分である。24 行目までは concat しているだけである。ただし、それらをすべて付加された環境も一緒に返している。その後、それを表示していくのだが、let 宣言の列で同じ変数に複数回束縛している場合は、一番最後の束縛の結果のみを表示する必要があるので、変数名を見ながら、要らないものは削除している。let 式については、eval.ml の 117〜129 行目にあるが、and でつながれたものしかないので、ループが一つで済んですっきりしている。

2.7 Exercise 3.3.3

2.7.1 設計方針

シンプルにやろうとして上手くいかなかったので、ファイルの中身を全部 string にして、;; で区切ってリストにして順番に lexer に与えていくことにする。また、ファイル中のプログラムにの実行中にエラーがあった時は、そのプログラムは実行は中断するが、その後に他のプログラムがあった場合、それらは実行される。

2.7.2 実装の詳細

まず、main.ml の 139 行目以降で、ファイル名が与えられているならファイル用の read-eval-print ループへ、そうでないなら通常の read-eval-print

ループへ飛ぶようにしている。本体は 54〜128 行目であるが、81 行目以降は通常のものと大差ない。関数を上から見ていくと、まず、ファイルを開いて、空文字列への参照 str を作っている。そして、ファイルの中身を一行ごとに空白で区切りながら str の中身と結合していく。ファイルを読み終えるとファイルを閉じて、str の中身を;; で分割してリストにしていく、これは get_str_list_by_semisemi 関数で行っており、一見複雑だが、要するに左から見ていって;; にであったところで切り出しているだけである。なお、;; も一緒に切り出さないと、parser がプログラムと認識してくれないのでそうしている。そうしてできたリストの中身を順に lexer に渡していくことでファイルの中身を実行している。リストが空になると、ファイルの実行が終わったことを表示して、通常の read-eval-print ループへ移行する。

2.8 Exercise 3.4.1

教科書通りにしただけなので、設計方針等はない。教科書に載っていない parser.mly の FunExpr の部分も特筆することはないほど単純である。pp_val についても、<fun> と表示するようにしただけである。テストプログラムは以下である。

```
# let y = 10;;
val y = 10
# let apply = fun f -> fun x -> f (x + y);;
val apply = <fun>
# let y = 5;;
val y = 5
# apply (fun x -> 2 * x) 10;;
val - = 40
```

これにより、高階関数がきちんと動作していること、関数定義時にきちんと関数閉包が作られていることがわかる。

2.9 Exercise 3.4.2

2.9.1 設計方針

(演算子) を認識する新たな文法を parser に追加する。当然これを関数適用に使いたいので優先度は AppExpr と AExpr の間とする。アクションは単に FunExp を返すだけでよい。

2.9.2 実装の詳細

parser.mly の 104〜112 行目に FunInfixExpr という新たな非終端記号を追加した。それぞれ、仮引数の名前を適当に決めて、その演算を行うだけの関数を返す。

2.10 Exercise 3.4.3

2.10.1 設計方針

複数引数の関数は、カリー化の考えで、1引数関数に帰着できるので、parser.mly だけをいじればよい。上の方は単に parser において引数の部分をリストで受け取るようにして、アクション部でそれを展開すればよい。下の方はあくまで let による表現だとして LetAndExpr を拡張するような形で実装する。

2.10.2 実装の詳細

上の方は parser.mly の 150〜156 行目の部分である。パラメータを nonempty_list 関数を通してもらうことで、1 つ以上受け取ることができる。アクション部も、単に FunExp の中に FunExp ができるようにしてカリー化を実現しているだけである。下の方は parser.mly の 142〜148 行目に LetFunExpr というものを追加した。中身は前者とほぼ同じである。これを LetAndExpr で Expr の代わりに受け取れるようにしておけばよい。

2.11 Exercise 3.4.5

2.11.1 設計方針

「dfun」を予約語にして、parser の部分では通常の関数と同じように処理すればよい。eval では ProcV から環境を取り除いた DProcV を追加して、AppExp の評価時に関数が DProcV の場合は今の環境にパラメータの束縛を与えて本体を評価すればよい。

2.11.2 実装の詳細

設計の方針をそのまま単純に実装しただけで、あまり説明することはない。一つ説明するなら、DProcV の pp_val の結果は <dfun> にした。

2.12 Exercise 3.5.1

教科書通りにしただけなので、設計方針等はない。実装するときに教科書そのままでは何か問題が起こった気がするが、覚えていない。テストプログラムは以下である。

```
# let rec fact n =
    if n < 1 then 1
    else n * (fact (n - 1));;
val fact = <fun>
# fact 5;;
val - = 120
```

```
# let rec sum n =
    if n < 1 then 0
    else n + (sum (n - 1))
  in
    sum 10;;
val - = 55
```

これにより、let rec 宣言、式ともに正しく動作していることがわかる。

2.13 Exercise 3.5.2

2.13.1 設計方針

Exercise 3.3.2 と 3.3.4 の場合と同じように実装した。ただし、and に関しては振る舞いが異なる。and でつながれた let rec で束縛する ProcV の中の環境への参照が同じ環境を指している必要があるので、単純にループで同じ環境を共有していくことにした。

2.13.2 実装の詳細

parser は Exercise 3.3.2 と 3.3.4 の場合とほぼ同じなので割愛する。eval についても評価する環境と束縛を与える環境が同じになったので、むしろ単純である。同じ環境を共有するとか考えている内に、どうせ上書きされるのでわざわざ dummyenv を作らなくてもよいことに気づいたので、env をそのまま使いまわしている。

2.13.3 テストプログラム

テストプログラムは以下である。これは Objective Caml 入門に載っていた相互再帰関数の例である。

```
# let rec even n =
    if n = 0 then true else odd(n - 1)
  and odd n =
    if n = 0 then false else even(n - 1);;
val even = <fun>
val odd = <fun>
# even 6;;
val - = true
# odd 14;;
val - = false
```

きちんと動作していることがわかる。

2.14 Exercise 3.6.1

match 式については 2.16 節で述べる。

2.14.1 設計方針

まず、exp にリストを表すコンストラクタを作る。そして、新たな型として listExp というものを定義する。listExp は二つのコンストラクタを持っていて、一つは空リストを表し、もう一つは先頭の要素と残りのリストの組を表す。これを exp のリストのコンストラクタに持たせておく。同様のことを exval についても行っておく。lexer と parser の部分は単純に実装すればよい。eval については評価そのものは簡単である (exp と exval が同じような形を持つようにしたので)。::の演算も単純に実装すればよくて、一番複雑なのは pp_val するために string に変換する部分である。これは中身の要素を順に見ていく形で実装した。

2.14.2 実装の詳細

syntax.ml の 32 行目と 41 行目が関係している。設計方針で述べた通りに実装している。lexer については、「[]」で一つのトークンを作ろうと思ったが、次の課題を踏まえて左かっこと右かっこそれぞれがトークンを作るようにした。parser については、空リストは AExpr の一部として、ConsExpr は MExpr より弱く、CmpExpr よりは強くした。eval は eval.ml の 13、14 行目にリストに対応する値を追加した。23〜38 行目の string_of_list 関数で ListV を string に変換している。この関数は、まず空リストなら「[]」に変換する。そうでなければ、「; arg1; arg2; ...]」の形を作って、最後に頭二文字を削って「[」を結合する。評価の部分は 157〜165 行目だが、単純に要素を順番に評価しながら対応する ListV を構築していく。

2.15 Exercise 3.6.2

2.15.1 設計方針

parser をいじるだけでよい。AExpr に追加するような形で実装する。始まりの「[hoge;」があって「fuga;」で続いていくか「]」で終わるかのどちらかであると考えて実装する。

2.15.2 実装の詳細

parser.mly の 123〜128 行目が関係している。設計方針で述べた通りである。ここで返すのは Cons であって、ListExp は AExpr が返してくれる。

2.16 Exercise 3.6.3/Exercise 3.6.4

2.16.1 設計方針

少しずつ拡張していくようにしたがこれは失敗で、新しい機能を追加するたびに元のプログラムのほとんどを変更していた。完成したものに対する設計方針を述べる。まず、parser にパターンに関する文法を追加する。そして、eval にパターンマッチする関数を作っておいて、match 式の評価に利用する。match 式に対応する exp のコンストラクタはマッチされる式のリストと、パターンのリストと本体式の組のリストを持つ。eval では、マッチされる式をまず順番に評価して、値のリストを得た後に、パターンマッチをして、マッチすれば本体式を評価する。また、各パターンで同一の変数が使われているかをチェックする。さらに、ワイルドカードも実装する。

2.16.2 実装の詳細

考え方は syntaxc.ml の 35〜39 行目のコメントを見ていただけるとわかると思う。parser は parser.mly の 177 行目以降が関係している。一つ以上を許すものが多いので、More... という導出規則が多くて見づらくなっているが、そこを無視すればわかりやすくなる。APattern というのは AExpr のようなもので、最も単純なパターンである。Pattern というのはリストに関するパターンを記述できるようにしたものである。PatternMatchExpr というのは、(パターン) -> (本体式) という部分を表している。MatchExpr は match 文全体を表している。次に、eval.ml の 50〜62 行目の pattern_match 関数について説明する。これは、パターンのリストとマッチされる式のリストを受け取って、マッチしたなら束縛すべき変数と値の組のリスト、マッチしなかったなら MatchError というエラーを投げる関数である。マッチしているかどうかの判断は単純に二つのリストを頭から順に見ていっているだけである。ただし、パターンのリストの型は exp list で、マッチされる式のリストの型は exval list であることには気をつける必要がある。実際に評価する部分は eval.ml の 166〜209 行目で、少しコメントを付けてあるので、ここで説明するよりもそちらを見ていただいた方がわかりやすいと思う。and でたくさん関数が宣言してあるが、他の関数を呼び出しているのは outer_loop 関数だけなので、実はあまり and でつなぐ意味はなく、トップダウンに考えて実装したことが表れているだけである。

2.17 Exercise 3.6.5

2.17.1 設計方針

各二項演算の右側に、「できるだけ右に延ばして読む」構文が来ても良いようにを拡張する。そのために、「できるだけ右に延ばして読む」構文を一つにまとめる。そして、それを各二項演算の右側に追加する。

2.17.2 実装の詳細

parser.mly の 58〜64 行目に「できるだけ右に延ばして読む」構文をまとめた LookRightExpr を追加した。そして、各二項演算の右側にそれを追加しただけである。

3 感想

難しい課題と簡単な課題の差が大きいと思った。特にパターンマッチの実装はすごく難しかった。その結果、実装がちょっと複雑になったので、このレポートで説明することを放棄してしまった。ただ、Exercise 3.6.4 は、一般的なパターンマッチという言葉が抽象的、かつ、Exercise 3.6.1 で実装したパターンマッチと ocaml のパターンマッチの差が大きい、ということで何を実装すればよいのかがわかりづらいのはどうなのかと思った。実際、ワイルドカードを実装した後と前で通ったテストの数が変わらなかったりした (他の機能がなかったせいかもしれないが)。ともあれ、インタプリタの作成は、プロセッサの作成よりもずっと面白いと感じた。