レポート 2

提出期限:2019/6/14
提出日: 2019 年 7 月 4 日

1029293806　大山 偉永

0.1 Exercise3.2.1

ML1 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.

ML1 インタプリタの実装は与えられた例のコードをコピーすれば実行できる。プログラムの説明はコード内に記す。cui.ml の環境をメソッド extend によって拡張子 ii に 2、iii に 3、iv に 4 を束縛させた。実際にインタプリタを立ち上げて ii+iii+iv を実行すると
$val\ -\ =\ 9$ が返ってくる。

0.2 Exercise3.2.2

不正な入力を与え 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.

cui.ml において大域環境を読み込み入力された値を評価する部分を関数として定義し、その間にエラーが発生すれば try-with 文により特定のメッセージを返し再び、その定義した関数を呼び出すように設計した。なおエラー文はひとつでも可能だが少し調べて 3 つに分けて出力できるようにした。特に Eval でのエラーをそのままインタプリタ上に表示するようにしたことで自分のデバッグもどこのエラーなのか判断しやすくなりいい改良だった。

0.3 Exercise3.2.3

論理値演算のための二項演算子 &&, || を追加せよ.

まず parser.mly において他の加算や乗算と同様に AND と OR の token を宣言する。次にこれらの表現を定義する ANDExpr と ORExpr を実装する。ここで注意が必要なのが演算子&&と || において&&のほうが優先順位が高いため ORExpr が ANDExpr を内包するような形で定義する。そして&&より比較演算子のほうが演算適用優先順位が高いため比較演算を表す LTExpr を ANDExpr が内包するように書く。syntax.ml では抽象構文木のデータ型を定義する。&&と || に関してはすでに定義されているヴァリアント型 BinOp に or と and の型を付加する。最後に解釈部 eval.ml で構文木を評価する。プリミティブ演算を適用する部分の関数 apply_prim に変更を加える。この関数の引数 op が syntax で定義した or 型か and 型かまたそれ以外かをパターンマッチで場合分けし評価する。ここで実際に ocaml の&&,|| 演算子を用いてこのプリミティブ演算の評価を決定する。ここでテストケースの true||undef もしくは false&&under はそれぞれ true と false を出力するように定義する必要があるのでこの場合に関しては関数 eval_exp ないで評価する。評

価する exp 型の値が BinOp (op, exp1, exp2) 型でかつそのときの op が or または and の時で exp1 もしくは exp2 が true もしくは false の時にそれぞれ true と false を出力するように評価の決定を行う。ここでこのケースに対応しない場合、未定義の undef 型が評価されてしまい undef が unbound value としてエラーが出てしまう。こうすることで&&と || の実装が完了した。

0.4 Exercise3.2.4

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．

これは lexer を変えるだけで実装が終わる。字句解析において main ルールで (*が現れた時はヒントにあるように新たに自分で comment ルールを定義しそこに飛ぶようにする。ここでこの字句解析のルールは引数を取ることができ、はじめこの comment ルールに飛んだ時はその引数を 0 とする。この comment ルール内でさらに字句"(*"が現れた時は comment 1 を呼び出す。このようにコメント内で (*が現れるたびに引数を一つ大きくした comment ルールを再帰的に呼び出す。逆に*) が現れた時はそのときの引数-1 した引数の comment ルールを呼び出しその引数-1 が 0 になったときにもとの main ルールに戻るように定義する。

0.5 Exercise3.3.1

ML2 インタプリタを作成し，テストせよ．

与えられたコードを所定の一にコピーアンドペーストすれば実行できる。プログラムの説明はコード内に記す。

0.6 Exercise3.3.2

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．

parser.mly の文法規則 toplevel のなかに LET x=ID EQ e=Expr top=toplevel {VarDecl(x,e,top)} を追加する。こうすることで上で追加した文法規則の toplevel の部分が文法規則 toplevel の中で別に定義されている文法規則 LET x=ID EQ e=Expr SEMISEMI { Decl (x, e) } を呼び出すことができ、let a = 3 let b = 4;; のような定義もすることができるようになる。ここで LET x=ID EQ e=Expr top=toplevel {VarDecl(x,e,top)} のように書いてしまうことで構文解析では let a = 3 5 や let b = 4 true のような表現も通ってしまうが eval.ml での構文木解釈時にエラーを出力するので実用上は問題ない。この 新たに追加した文法規則のデータ型を program 型の VarDecl として id 型、exp 型、program 型の値の情報を持つように syntax.ml で定義する。eval.ml の解釈部においては eval_decl でこの宣言を評価できるようにする。VarDecl とともに VarDecl (id ,e, top) の top の部分も program 型なのでまず exp 型の e を eval_exp で評価してその値を id で束縛

し環境に追加、その更新された環境で再び再帰的に eval_decl を呼び出して top を評価する。こうすることで let a = 3 let b = 4;; のような式が正しく評価される。

0.7 Exercise3.3.4

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.

exercise3.3.2 のように let を 2 つ連続で並べるだけの時はひとつ目の宣言を環境に追加してよかったため簡単だったが今回は and で結ばれた宣言は直前の宣言の束縛を参照しないようにする必要があり、さらに and で結ばれたすべての宣言が終わったあとにそれら一連の宣言の変数の束縛を環境に追加する必要がある。またこの問では宣言と同時に in も用いれるようにする必要があるため、parser.mly での文法規則の定義も難しかった。まず parser.mly において in 以下のない let and let のみの宣言の文法規則を toplevel の下に追加する。これは LET x=ID EQ e1=Expr LETAND e2=LetAndExpr SEMISEMI { LetAndDecl(x, e1, e2) } のように宣言する。このデータ型 LetAndDecl (x, e1, e2) は syntax.ml で id 型、exp 型、program 型の値を持つ program 型の中の LetAndDecl 型として定義する。また上の追加した文法規則の中の LetAndExpr は新たに定義した文法規則でありこの let - and let - の文を実現する文法規則を定義する。次に let and の式に in がついた文のための文法規則を LetAndInExp : LET x=ID EQ e1=Expr LETAND e2=LetAndExpr IN e3=Expr { LetAndInExp((LetAndRecExp(x,e1,e2)),e3) } として定義する。上と同様に LetAndInExp を program 型と exp 型の値を持つ exp 型のデータ型として syntax.ml で定義する。これらで定義した構文を eval.ml で解釈する。まず LetAndDecl は program 型なので eval_decl 関数の中で最初の let 文を評価し変数の束縛を実行するが、その束縛は環境には加えず最初の与えられた環境でふたつ目の and 以下の let 宣言を再帰的に再び eval_decl で評価する。この評価で返ってきた環境に最後に最初の let 宣言の束縛を加える。こうすることで各々の let 宣言は互いの宣言の束縛に影響を与えずに評価できなおかつ最後にそれらの宣言の束縛を環境に追加することができる。LetAndInExp((LetAndRecExp(x,e1,e2)),e3) の評価についてはこれは exp 型なので eval_exp 関数で評価を行うがこの LetAndInExp 型がもつひとつ目のデータ型 LetAndRecExp(x,e1,e2) は eval_decl で評価しそこで返された環境の中で e3 を eval_exp 関数で再帰的に評価する。結局 eval_decl 関数の中でも eval_exp 関数を呼び出し eva_exp 関数の中でも eval_decl 関数を呼び出しているため相互再帰となっているがこうすることで let a=1 and b=2 in a+b のような式を実現できる。let and の式ないで同じ識別子に束縛そしようとする文が現れたらエラーを返す必要があるが、これに関しては eval.ml 内でリストの参照を作っておき、let and で識別子が束縛されるたびにその識別子の値をそのリストに追加する。そして let and の再帰的に評価していく中でその評価する式の中の識別子を先ほど定義したリストで検索しもしそのリストに宣言された識別子が含まれていたらエラーを返すようにした。

0.8 Exercise3.4.1

ML3 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.

まずは資料のコードをコピーするが、parser.mly において文法規則 FunExpr が未定義なので自分で定義する必要がある。function 文は fun x -¿ x+1 のようなかたで書かれるので FunExpr は FUN e1=ID RARROW e2=Expr { FunExp (e1, e2) } のような形で定義できる。それ以外はすでにテキストのコード内で実装済みである。

0.9 Exercise3.4.2

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.

問題分例文から (+) のような文字列を関数として認識する必要があることが読み取れる。つまりこの構文が来たら ProcV(_,_,_) を返す必要があるとわかる。実装についはまずここまでの問と同様にして Parser.mly で文法規則 LPAREN PLUS RPAREN { MidPlusExp } と LPAREN MULT RPAREN { MidMultExp } を AExpr に追加する。こうすることでこの (*) などを関数として扱った際にこの表現も普通の関数と同様に AppExpr で適用できるようになる。次にこの MidMultExp と MidPlusExp を exp 型のデータ型として syntax.ml で定義する。最後に eval.ml で eval_exp の引数がこの型の時に引数を持ちその和（積）を返す関数を ProcV 型で返すようにする。実際には適当に id を”x”として ProcV の id とし、この関数閉包ボディの部分は引数 y で x+y を返すファンクションとして記した。このようにすることで実装が完了する。

0.10 Exercise3.4.4

加算を繰り返して 4 による掛け算を実現している ML3 プログ ラムを改造して，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 x*(maker maker (x + -1)) in
let times4 = fun x -¿ makemult makemult x in
times4 3
```

もとの 4 の掛け算を実行するプログラムが times4 の引数の数だけ makemult makemult a を呼び出してそのたびに 4 を加算していくために 4 の乗算が実行されていたことを理解すると、4 を加算していた部分のかわりに引数の値を乗算すればいいことがわかる。そうすることで引数の回数だけ

再帰的に関数が呼びだされ階乗が計算される。

0.11 Exercise3.4.5

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．

fun と dfun で実装が違う部分は解釈部の eval.exp だけなので parser.mly と syntax.ml は fun の実装と同様にするだけで良い。dfun の評価については関数が宣言された際の環境の情報を保持しておく必要がないので新たに環境の情報を保持しない環境閉包 DprocV を eval.ml で定義し DFunExp (e1, e2) を eval_exp で評価するときは DProcV を返す。関数適用時は AppExp(exp1,exp2) のなかで exp1 が ProxV の場合と DProcV の場合でパターンマッチを行い DProcV(id,body) の時は id を exp2 の評価結果で束縛しこの関数適用が呼びだされた時の環境を拡張して body を評価する。以上でこの dfun の実装は完了する。

0.12 Exercise3.4.6

以 下のプログラムで， 二箇所の fun を dfun (Exercise3.4.5 を参照) 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
val - = 25
最後 fact 5 が呼び出されると直前の fact が 5 に適用されるが、関数閉包 ProcV(id,body,env) の id を 5 で束縛し body を評価する際、この body 内の fact はこの関数宣言時の環境にあった let fact = fun n → n + 1 を参照するため 5*5 が行われ 25 という結果が出力される。

let fact = dfun n -¿ n + 1 in let fact = fun n -¿ if n ¡ 1 then 1 else n * fact (n + -1) in fact 5
val - = 25
上のプログラムと違い、ひとつ目の fact の宣言が dfun になったがこの fact 関数はこれが宣言された時の環境に依存しない関数なのでこの変更はひとつ目のプログラムに変化を与えない。

let fact = fun n -¿ n + 1 in let fact = dfun n -¿ if n ¡ 1 then 1 else n * fact (n + -1) in fact 5
val - = 120
最後 fact 5 が呼び出されると直前の fact が 5 に適用されるが、ふたつ目の fact は dfun で宣言されているためその関数閉包 DProcV(id,body) は宣言時の環境の情報を保持しない。したがってこ

の関数閉包の id を 5 で束縛し body を評価する際の環境には評価時の環境が適用されるため、この body 内の fact は現在の fact が再帰的に呼びだされ階乗を計算できるようになる。

let fact = dfun n -¿ n + 1 in let fact = dfun n -¿ if n ¡ 1 then 1 else n * fact (n + -1) in fact 5
val - = 120
ふたつ目のプログラムと同様でひとつ目の fact の宣言の dfun 化は出力に影響を及ぼさない。

0.13 Exercise3.5.1

まず lexer.mll の予約語のところに REC を追加する。次に parser.mly においてまず token として REC を宣言する。toplevel のところに let rec の宣言の文法規則 LET REC f=ID EQ FUN x=ID RARROW e=Expr SEMISEMI {RecDecl(f,x,e)} を追加する。また宣言以外で用いいられる let rec として LetExpr のなかにも LET REC x=ID EQ FUN y=ID RARROW e1=Expr IN e2=Expr { LetRecExp (x, y, e1, e2) } の文法規則を追加する。eval_exp は資料中のコードで実装済みであるので最後に eval_decl の引数が RecDecl(f,x,e) の場合を実装する。これは eval_exp とほぼ同じで最後に宣言された再帰関数の識別子とあらたにこの関数の情報を加えた環境とこの関数閉包を返せばよい。以上で再帰関数の実装は終了する。

0.14 Exercise3.6.5

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.

最初の定義では IfExpr は文法規則 Expr の中にあったが、この IfExpr を AeExpr の中に入れることで最も優先順位が高い文法規則となりカッコがなくてもこの if 節を一つの塊として優先的に解釈してくれる。同様に let,fun 文も AEExpr の中に入れることでこの文法規則が優先的に読み出されかっこがなくても使えるようになる。

0.15 感想

非常に課題の量が多いと思った。必須課題と星 5 個を終わらすのに対して、発展問題をとききるの難しすぎた。また回答？模範実装例みたいなのがあったら嬉しいと思った。引き続き型推論も頑張っていきたい。