計算機科学実験演習３　ソフトウェア　レポート２

氏名：吉村仁志

提出日：７月４日（木）

第1章　ML[1] インタプリタ

1.1　Exercise　3.2.1

この問題は cui.ml の initial_env を iv、iii、ii を IntV 4、IntV 3、IntV 2 に束縛して拡張すれば良い。環境の拡張には Environment.extend 関数を使う。実行結果は 10 になる。

1.2　Exercise　3.2.2

字句解析と構文解析は cui.ml の decl を求める際に行われる。また、実際に構文解析結果を計算するのは、eval.ml であり、cui.ml の eval_decl の部分で計算が始まる。エラーが発生する可能性がある部分はこれらになるので、read_eval_print の let decl = 以降を try-with 構文で囲めばエラー発生時の挙動を with 以下で処理できる。また、lexer.mll、parser.mly、eval.ml で exception Error を定義しておき、with 以降でそれぞれのエラーにマッチさせ、エラーメッセージを出力する。また、再びプロンプトに戻るためにそれぞれのエラーが発生したとき、発生していないときすべての場合で read_eval_print を再帰呼び出しする。

1.3　Exercise　3.2.3

lexer を、&&を AAND、|| を OOR で拡張、binOp を AAND、OOR で拡張する。apply_prim の拡張だが、AAND の評価結果を (exp1 && exp2)、OOR の評価結果を (exp1 || exp2) にする。ただし、exp1、exp2 は左側のオペランド、右側のオペランドに対応する。次に、true || undef、false && undef の２つについて考える。これらは、左のオペランドを評価したときに、右側の評価結果は全体の評価結果に関係ない。この場合、左側の評価を終えたときに右側を評価せずに評価を終了する。&&、|| にのみマッチする構文 BExpr を parser に追加する。これにマッチした場合、ANDORBinOp of binop * exp * exp が属性になる。次に eval での ANDORBinOp の評価について説明する。まず、binop は AAND か OOR のいずれかになるようにしている (parser の還元時アクションにこの２つしか出てこない)。左のオペランドを評価し arga に束縛する。次に、binop が AAND かつ arga が false のとき false、binOp が

OOR かつ arga が true のとき true を返す。そうでないときは、eval_exp env (BinOp(op, exp1, exp2)) を評価結果にする。

1.4 Exercise 3.2.4

まず全体の方針を説明する。lexer に (*、*) を追加する。規則 comment を lexer に追加する。また、(*の字句を読み取ったとき、以降の字句列の読み取りを comment に入るように comment lexbuf を呼び出す。また、(*、*) の数を cnt としてカウントする。前者の場合がインクリメントし、後者の場合はデクリメントする。コメントとして成立する場合は、字句列を左から cnt の値を計算したときに、常に 0 以上であり、最終的に 0 になることである。そこで、cnt を整数の参照として定義し、0 で初期化する。具体的に実装の中身について説明する。まず、main で (*を読み取ったときに!cnt をインクリメントし、comment lexbuf を呼び出す。*) に入った場合は、!cnt が-1 になるので、当然エラーになる。規則 comment では、(*を読み取ったときには!cnt をインクリメントし、comment lexbuf を呼び出す。*) を読み取ったときは、!cnt をデクリメントする。!cnt が 0 のときはコメントは終わるので、main lexbuf を呼び出す。!cnt が 0 より大きいときは、まだコメントは続くので、comment lexbuf を呼び出す。!cnt が 0 より小さいときは、エラーになる。

第2章　ML^2インタプリタ

2.1　Exercise　3.4.1

ML^2 インタプリタの作成だが、拡張する部分はすべて chap03-2.pdf で解説されており、特に説明する部分はない。

2.2　Exercise　3.4.2

まず構文解析の方針としては、構文解析で let ＜識別子＞ = ＜式＞ の各宣言を再帰的にマッチするようにする。そこで、parser に DecLetExpr を追加する。実際には、

$$\begin{aligned} &LET\ x = ID\ EQ\ e1 = Expr \\ &\qquad e2 = DecLetExpr\ \{\ DecDecl\ (x,\ e1,\ e2)\ \} \end{aligned} \tag{2.1}$$

のように e2 で再帰呼び出しをしている。また、最後には LetExpr にマッチするので、e=LetExpr を DecletExpr に追加する。DecDecl について説明する。定義は DecDecl of id * exp * program になる。これはまず、id を exp の評価結果で束縛し、e2 の部分は DecDecl または Decl になるので program 型である。構文解析結果は、

$$DecDecl(DecDecl(...Decl()..) \tag{2.2}$$

のようなネスト構造になる。また、eval.ml での評価について説明する。DecDecl は program 型であるので、eval_decl で処理する。DecDecl (id, e1, e2) を評価するとする。まず、e1 を評価し id に束縛し、環境をそれで拡張した新たな環境 newenv を作る。その上で eval_decl e2 newenv として再帰呼出しする。

2.3　Exercise　3.4.4

まず構文解析の方針としては、Exercise 3.4.2 と同じように

$$let\ <\text{識別子}> = <\text{式}> \tag{2.3}$$

の各宣言を再帰的に解析するようにする。そこで、parser に AndExpr を追加する。また、これはプログラムになるのでまず toplevel でマッチするようにする。toplevel に

$$\begin{aligned} LET\ x = & ID\ EQ\ e1 = Expr\ AND \\ & e2 = AndExpr\ SEMISEMI\ \{\ AndLet\ (x,\ e1,\ e2)\ \} \end{aligned} \tag{2.4}$$

を追加する。これにより and 以下の構文の解析に移る。AndLet について説明する。定義は AndLet of id * exp * program になる。これは、exp を評価した結果を id に束縛し、e2 は AndLet 、Decl になる。AndExpr は、構造は DecLetExpr と全く同じである。構文解析結果は、AndLet(AndLet(...Decl()...) のようなネスト構造になる。また、IN による式を定義することもできるため、それも実装する。これのための構文規則 LetAndInExpr を追加する。まず、構文解析を以下のように 3 つの部分に分けて考える。

$$let\ a\ =\ 1\ and\ ... \tag{2.5}$$

$$b\ =\ 1\ and\ ... \tag{2.6}$$

$$\begin{array}{c} \cdot \\ \cdot \\ \cdot \end{array}$$

$$c\ =\ 1\ in\ a + b + c \tag{2.7}$$

上から 2 つの還元時アクションは LetAndInExp、最後のそれは LetEndInExp になる。また、... の部分で LetAndInExpr の再帰定義をする。すなわち、(2.5)、(2.6)、(2.7) はそれぞれ下の (2.8)、(2.9)、(2.10) にマッチする。

$$\begin{aligned} LET\ x = & ID\ EQ\ e1 = Expr\ AND \\ & e2 = LetAndInExpr\ \{\ LetAndInExp\ (x,\ e1,\ e2)\ \} \end{aligned} \tag{2.8}$$

$$\begin{aligned} x = & ID\ EQ\ e1 = Expr\ AND \\ & e2 = LetAndInExpr\ \{\ LetAndInExp\ (x,\ e1,\ e2)\ \} \end{aligned} \tag{2.9}$$

$$\begin{aligned} x = & ID\ EQ\ e1 = Expr\ IN \\ & e2 = Expr\ \{\ LetEndInExp\ (x,\ e1,\ e2)\ \} \end{aligned} \tag{2.10}$$

構文解析結果は、

$$LetAndInExp(LetAndInExp(...LetEndInExp()...) \tag{2.11}$$

のようなネスト構造になる。eval.ml について説明する。まず and による並列宣言では、宣言が終わると同時に変数の束縛が一斉に行われる。また、一斉に束縛が行われるため、and による同じ名前の変数の束縛はできない。それの

実装を説明する。まず、先の事実から AndLet を評価するごとに環境を更新することはできないので、グローバル変数としてリストの参照 andletlist (id * exp) list を定義しておき、宣言されるごとにこれに追加するようにする。また、最終的に一斉に環境を更新するための関数 andlistadd 、新たに束縛しようとしている変数と同じ名前の変数がすでに宣言されていないかを判定する関数 findid を定義しておく。前者は、リストと環境を引数として受け取って、環境をリストで拡張しその環境を返すような関数である。後者は、id とリストを受け取って、リストを走査し id と同じ名前の変数があれば true 、そうでなければ false を返すような関数である。次に AndLet (id, e1, e2) の処理を説明する。まず findid 関数でいま定義しようとしている変数が andletlist に入っていないかを判定し、true のときにエラーを起こす。結果が false のもとで、id と e1 の評価結果のタプルを andletlist に追加し、同じ環境のもとで eval_decl env e2 env2 として eval_decl を再帰呼び出しする。また、最後に評価する Decl (id, e) に対してだが、ここで anslistadd を用いて環境を拡張するように Decl に機能を追加する。次に LetAndInExp 、LetEndInExp の評価について説明する。LetAndInExp (id, e1, e2) の評価についてだが、構造としては AndLet の評価方法と全く同じである。すなわち、findid で判定し、false のもとで新しい変数を andletlist に追加し、同じ環境のもとで eval_exp env e2 として eval_exp を再帰呼び出しする。LetEndInExp (id, e1, e2) の評価は、LetAndInExp と比較すると、andlistadd の拡張をせずに、andletlist で環境を拡張するように変更したものとなる。

第3章　ML^3インタプリタ

3.1　Exercise 3.4.1

chap03-3.pdf で解説されていない部分について解説する。解説されていない機能は FunExp に還元される構文規則である。それを FunExpr とする。式は、

$$fun\ x \rightarrow\ x + 1 \tag{3.1}$$

のようになるため、それを素直に構文規則にすれば良く、下のようになる。

$$FUN\ x = ID\ RARROW\ e = Expr\ \{\ FunExp\ (x,\ e)\ \} \tag{3.2}$$

また、FunExpr は式であるため Expr に e=FunExpr { e } を追加する。

3.2　Exercise 3.4.2

まず (+)、(*)、を字句解析する必要があるため、lexer に (+) を FPLUS、(*) を FMULT として追加する。次に構文解析を解説する。まず、(+) の単純な計算と、カリー化関数を定義することを考える。この構文規則として FPlusFunExpr を追加する。単純な計算の構文は、

$$FPLUS\ e1 = AExpr\ e2 = AExpr\ \{\ BinOp\ (Plus,\ e1,\ e2)\ \} \tag{3.3}$$

のように単なる足し算と同じである。次にカリー化関数を考えるが、そのために新しい式として FplmuFunExp of binop * exp * id を定義する。引数が１個の場合の構文は、

$$FPLUS\ e = AExpr\ \{\ FplmuFunExp\ (Plus,\ e,\ \text{"-"})\ \} \tag{3.4}$$

引数が０個の場合の構文は、

$$FPLUS\ \{\ FplmuFunExp\ (Plus,\ ILit\ 0,\ \text{"- -"})\ \} \tag{3.5}$$

のようにすれば良い。(FplmuFunExp の具体的な説明は後述する。) 次に関数適用を考える。関数適用として AppExp の構文を

$$e1 = AppExpr\ e2 = FPlusFunExpr\ \{\ AppExp\ (e1,\ e2)\ \} \tag{3.6}$$

などと変更する規則がまず考えられるが、これだけでは FPLUS が後続の引数にすぐに適用されてしまい、関数適用の左結合が実現できない。そこで AppExp に、左結合を実現できるように次の規則を追加する。

$$\begin{aligned} e1 =& AppExpr\ FPLUS\ e2 = AExpr\ \{\ AppExp \\ & (AppExp(e1,\ FplmuFunExp\ (Plus,\ ILit\ 0,\ \text{"- -"})),\ e2)\ \} \end{aligned} \tag{3.7}$$

すなわち、FPLUS に引数が後続しているときに還元時アクションで e1 を FPLUS 単体に適用するようにする。これにより左結合が実現できる。次に eval.ml での FplmuFunExp の評価について説明する。まず 3 つめのフィールドの id だが、これは引数の個数で場合分けするためのセレクタの役割をしている (bool 型などにしても問題ない)。これは関数になるので、

$$fun\ a\ b \rightarrow a + b \tag{3.8}$$

とおなじ関数を定義したのと同等となるようにすれば良い。(3.8) を構文解析すると下のようになる。

$$FunExp(a,\ FunExp(b,\ BinOp(Plus,\ Var\ a,\ Var\ b))) \tag{3.9}$$

これを eval_exp で評価すると、

$$ProcV(a,\ FunExp(b,\ BinOp(Plus,\ Var\ a,\ Var\ b)),\ env) \tag{3.10}$$

となるので、FplmuFunExp の評価結果をこれと同じ形にすれば良い。そこで、ダミーの仮引数として id 型 "a"、"b" を用いて、引数が 0 個のとき、

$$ProcV(a,\ FunExp(b,\ FplmuBinOp(Plus,\ \text{"}a\text{"},\ \text{"}b\text{"})),\ env) \tag{3.11}$$

引数が 1 個のとき、

$$\begin{aligned} & ProcV(\text{"}b\text{"},\ FplmuBinOp(Plus,\ \text{"}a\text{"},\ \text{"}b\text{"}), \\ & \qquad ref\ (Environment.extend\ \ \text{"}a\text{"}\ arga\ env))) \end{aligned} \tag{3.12}$$

とすれば良い。FplmuBinOp についてだが、これは syntax で FplmuBinOp of binop * id * id として定義され、二つの id の名前の変数の値を足すような exp である。評価は BinOp とほぼ同じで容易である。また、引数が 1 個のときは "a" をその引数の評価結果で束縛したもので環境を拡張し、引数を一つ受け取って、それと "a" の値を足すような関数を返す。以上により、(+) に関する実装は完了する。(*) についても全く同様である。

3.3 Exercise 3.4.5

まず lexer に dfun のための予約語 DFUN を追加する。次に新しい exp 型 DfunExp、exval 型 DProcV を定義する。DfunExp は、parser の FunExpr

で dfun にマッチしたときに還元される。すなわち、

$$DFUN\ x = ID\ RARROW\ e = Expr\ \{\ DfunExp\ (x,\ e)\ \} \qquad (3.13)$$

次に DfunExp を eval.ml で評価すると、DProcV になる。また、DProcV のフィールドは ProcV と全く同じである。実際に関数適用をするときだが、AppExp を eval_exp で評価するとき、funval を ProcV とマッチするかどうかを判定するが、ここで DProcV のマッチも増やし、DProcV の評価を別に行う。次に動的束縛の実装を説明する。まず関数宣言時の環境 (DProcV から取り出した環境) を newenv 、今現在の環境 newenv2 をそれぞれ実引数の束縛で拡張する。動的束縛は新しい変数が優先されるので、両方共で同じ名前として宣言されている変数があるときは、newenv2 のそれを優先するようにする。すなわち、try 構文を利用してまず newenv2 のもとで body を評価し、成功したときはそれが求めるべき結果である。失敗したときは newenv のもとで body を評価すれば良い。これで動的束縛が実現できる。

3.4 Exercise 3.4.6

まず結果としては、2つめの fun が dfun のとき再帰関数として定義され、評価結果は 120 となり、そうでないときは、再帰関数として定義できず評価結果は 25 になる。fact の定義は以下のように 2 回行われている。

$$let\ fact\ =\ fun\ n\ \rightarrow\ n+1 \qquad (3.14)$$

$$let\ fact\ =\ fun\ n\ \rightarrow\ if\ n<1\ then\ 1\ else\ n\ *\ fact(n\ +\ -1) \qquad (3.15)$$

fact 5 の評価するときを考える。2つめの fun が fun のときには、(3.15) の定義の最後の fact の呼び出しでは、静的束縛により、(3.15) が宣言されたときの fact を呼び出すので、(3.14) の fact が利用される。その結果、fact (n + -1) の評価結果は n となり、fact 5 の結果は 5 * 5 = 25 となる。2つめの fun が dfun のときは、動的束縛により fact 5 の評価をするときの環境の fact を利用するので、(3.15) の fact を呼び出すことになり、再帰的に fact が呼び出され、結果は 120 になる。

第4章　ML^{4}インタプリタ

4.1　Exercise 3.5.1

chap03-4.pdf で解説されていない部分について説明する。まず rec の字句解析が必要だから、lexer に 予約語 REC を追加する。次に構文解析を説明する。まず let rec 宣言の構文解析だが、これはプログラム型であり、toplevel に以下の規則を追加する。

$$\begin{aligned} &LET\ REC\ x1 = ID\ EQ\ FUN\ x2 = ID\ RARROW \\ &\qquad e = Expr\ SEMISEMI\ \{\ RecDecl\ (x1,\ x2,\ e)\ \} \end{aligned} \tag{4.1}$$

次に let rec 式の構文解析だが、まず新しい規則 LetRecExpr を追加する。これは素直に下のようにマッチさせれば良い。

$$\begin{aligned} &LET\ REC\ x1 = ID\ EQ\ FUN\ x2 = ID\ RARROW \\ &\qquad e1 = Expr\ IN\ e2 = Expr\ \{\ LetRecExp\ (x1,\ x2,\ e1,\ e2)\ \} \end{aligned} \tag{4.2}$$

また、e=LetRecExpr を Expr に追加する。以上で構文解析は完了する。次に、RecDecl の評価について考える。これは、program 型であるので eval_decl で評価する。また LetRecExp の評価での dummyenv への破壊的代入までは全く同じであり、それで評価は終わりである。以上で ML^4 インタプリタの実装は完了する。

4.2　Exercise 3.5.2

最初に構文解析について説明する。まず式の構文解析を考える。let and の並列宣言と同じように、and 以下の再帰定義を行う。構文を下のように 3 つに分割する。

$$let\ rec\ f\ =\ fun\ n\ \rightarrow\ f\ (n + -1)\ and\ ... \tag{4.3}$$

$$g\ =\ fun\ n\ \rightarrow\ g\ (n + -1)\ and\ ... \tag{4.4}$$

・
・
・

$$h \ = \ fun\ n \ \rightarrow \ h\ (n + -1)\ in\ f\ 3 \tag{4.5}$$

これら 3 つの構文解析を考えるが、そのための新しい規則 RecAndInExpr を追加する。(4.3)、(4.4)、(4.5) はそれぞれ下の (4.6)、(4.7)、(4.8) にマッチするようになる。また、... の部分で再帰定義が行われる。

$$\begin{aligned} &LET\ REC\ x1 = ID\ EQ\ FUN\ x2 = ID\ RARROW\ e1 = Expr \\ &\quad AND\ e2 = RecAndInExpr\ \{\ LetRecExp\ (x1,\ x2,\ e1,\ e2)\ \} \end{aligned} \tag{4.6}$$

$$\begin{aligned} x1 =&ID\ EQ\ FUN\ x2 = ID\ RARROW\ e1 = Expr \\ &AND\ e2 = RecAndInExpr\ \{\ LetRecExp\ (x1,\ x2,\ e1,\ e2)\ \} \end{aligned} \tag{4.7}$$

$$x = ID\ EQ\ e1 = Expr\ IN\ e2 = Expr\ \{\ LetEndInExp\ (x,\ e1,\ e2)\ \} \tag{4.8}$$

構文解析の結果は、下のようなネスト構造になる。

$$LetRecExp(LetRecExp(...(LetEndInExp()...) \tag{4.9}$$

これにより、式の再帰関数の並列宣言が実行できる。次に、プログラムの再帰関数の実装を考える。構文解析は、(4.3)、(4.4)、(4.5) のうち、(4.5) の in 以下がない場合と同じである。またそれを (4.5)*とする。まず toplevel に (4.3) に相当する構文の規則を追加する。以下のようになる。(RecAndLet については後述する。)

$$\begin{aligned} &LET\ REC\ x1 = ID\ EQ\ FUN\ x2 = ID \\ &\qquad RARROW\ e1 = Expr\ AND\ e2 = RecAndExpr \\ &\qquad SEMISEMI\ \{\ RecAndLet\ (x1,\ x2,\ e1,\ e2)\ \} \end{aligned} \tag{4.10}$$

ここで、再帰的な規則になっている新しい規則、RecAndExpr が追加されている。これは、上の and 以下から移る規則であり、中身は (4.4)、(4.5)*に相当する構文にマッチする。(4.4)、(4.5)*はそれぞれ、下の (4.11)、(4.12) にマッチする。

$$\begin{aligned} x1 =&ID\ EQ\ FUN\ x2 = ID\ RARROW\ e1 = Expr \\ &AND\ e2 = RecAndExpr\ \{\ RecAndLet\ (x1,\ x2,\ e1,\ e2)\ \} \end{aligned} \tag{4.11}$$

$$\begin{aligned} &x1 = ID\ EQ\ FUN\ x2 = ID \\ &\qquad RARROW\ e = Expr\ \{\ RecDecl\ (x1,\ x2,\ e)\ \} \end{aligned} \tag{4.12}$$

構文解析の結果は、下のようなネスト構造になる。

$$RecAndLet(RecAndLet(...(RecDecl(...)) \tag{4.13}$$

これの評価について説明する。RecAndLet の定義は、RecAndLet of id * id * exp * program となる。これは、RecDecl に program 型のフィールドを追加し

たものである。すなわち、ネスト構造による後続のプログラム、RecAndLet、RecDecl が入ることになる。これは eval_decl で評価するが、RecDecl の評価での dummyenv への破壊的代入までは全く同じである。その後に、program 型のフィールドを eval_decl で再帰的に呼び出すことで、上のネスト構造を処理できる。次に相互再帰関数の定義を考える。まず下のような相互再帰を評価することを考える。(実際には、この式の評価は止まらないが簡単のために条件を省いている。)

$$let\ f\ =\ fun\ n\ \rightarrow\ g\ n\ and\ g\ =\ fun\ n\ \rightarrow\ f\ n \tag{4.14}$$

まず、現在の実装でこれを宣言後に評価すると、f の評価の際に g が見つからないと怒られる。すなわち、f の宣言時の環境に g が入っていないことが問題である。宣言が終了したときの環境には f、g の両方が入っているので、宣言が終了したときの環境で評価を行えば、正しく実装できる。具体的な実装では、今までは ProcV の body を評価するときには env' を利用するが、まず try 構文を利用し、env' のもとで、body を評価し、失敗したときは env のもとでもう一度評価する。これにより正しく実装できる。

第5章　その他

5.1　工夫した点

まず、実装にあたって工夫した点としては、構文解析のデバッグのために新しいファイル syntax_debug.ml を作ったことである。これは、構文解析の結果を eval.ml で評価を始まる前に出力するような関数が定義されてある。これにより、パーズでの予期しない解析がされている時の発見が容易になったり、構文解析結果を eval.ml でシミュレートしやすくなった。

5.2　感想

難しかった点として、ひとつ目は、まず ML^2 インタプリタの Exercise3.4.4 の and による並列宣言で、新しく束縛された変数を環境に一斉に追加するために、別の関数や変数をグローバルで定義したりと、複雑になった部分である。グローバル変数を定義などせずに、工夫次第でもっと簡潔に書けるのではと思った。ふたつ目は、ML^3 インタプリタの Exercise3.4.2 の中置演算子の関数適用が左結合できていない部分になかなか気づかずに、初めは parser に冗長な規則が増えてしまった部分である。ただ最終的に、個人的には割と簡潔になった気がしたので良かった。３つめは、ML^3 インタプリタの Exercise3.4.3 である。これは取り組んだのにうまく行かなかったものである。関数を宣言した時に、body が関数適用の場合にうまくパーズできなかった。おそらく構文解析に関する理解ができていない点が原因であるが、未だによくわからない。４つめは、構文解析のコンフリクトの発生を止められなかったことである。いまでもコンフリクトが存在し、それにより予期せぬ挙動を起こしている部分はほとんどない気はするが (Exercise3.4.3 には影響しているかもしれない)、出来る限り、似たような構文規則を作らないようにしても少ししか改善しなかった。構文解析についてより勉強が必要だと思った。