計算機科学実験 3SW レポート 2

学籍番号: 1029350386
氏名: 高橋奏太

2025 年 6 月 23 日

ex3.2.1(必修)

1 設計方針

cui.ml の中で、initial_env としてプログラム開始時の大域環境が定義されている。そのため、その定義式の中に ii,iii,iv を同じように追加する。

2 実装にあたり工夫した点

空の環境 empty に対して extend 関数で ii,iii,iv を 2,3,4 に束縛されるのを入れ子構造で作成した。

3 インタプリタ動作確認

```
# if v<0 then true else false;;
val - = false
# x + v * 3 + i;;
val - = 26
```

条件分岐、変数参照、op<、真偽値、加算乗算、整数の機能を確認している。

4 動作確認

```
# ii;;
val - : int = 2
# iii;;
```

```
val - : int = 3
# iv;;
val - : int = 4
# iv + iii * ii;;
val - : int = 10
```

※現在の実装したインタプリタだと型推論が実行されるが、教科書の流れに従った変更の場合、val - = 10 のように動作する。

ex3.2.3 *

1 設計方針

&&,||を追加した BNF に対応する抽象構文木を表す必要があるため、抽象構文木を表すデータ型を定義する syntax.ml、受け取った文字列をトークン列に変換する字句解析器を生成する lexer.mll に&&,||の定義を加える。

また、そのトークン列を受け取り構文木に変換する構文解析器を生成する parser.mly に、&&,||のトークンと、結合の強弱・結合性に基づいた文法規則を加える。

そして、それによってできた構文木を解釈するための eval.ml に演算子 And,Or を受け取った時に返す式の評価を加える。

2 実装にあたり工夫した点

parser.mly に ANDExpr,OORExpr を追加するとき、既存の演算子 < との結合の強弱を成立させるために Expr の文法規則を変更した点。

eval.ml で And,Or を評価する際、apply_prim 内で計算する仕様にした場合、false&&undef は短絡評価で結果が false に決まるはずが実際には計算できないのでエラーとなる。それを防ぐために、eval_exp で式を評価する段階で短絡評価のパターンを記述している。

3 動作確認

```
# 3<4 && true;;
val - : bool = true
# false || 4<0;;
val - : bool = false
```

※型推論が行われているが、教科書の流れに従った変更の場合、val -= false のように動作する。

ex3.3.1(必修)

1 設計方針

MiniML2 インタプリタを作成するにあたって、変数宣言の機能として let 宣言と let 式を追加するので、BNF を拡張して let 部分を追加する必要がある。

その拡張に対応するために、syntax.ml に構文木の拡張を、lexer.mll に予約語と記号の追加を、parser.mly に let は結合が if と同程度に弱いことを踏まえた構文規則の追加を、行う。

また、let 式を評価するために eval.ml にて、構文木を評価する関数とその関数を用いて評価結果などの組を返す関数に let に対応したパターンマッチを追加する。

2 インタプリタ動作確認

```
# let x = 1 in let y = 2 + 2 in x + y*v;;
val - : int = 21
# let x = true;;
val x : bool = true
```

let 式、let 宣言の機能を確認している。
※型推論が行われているが、教科書の流れに従った変更の場合、val - = 21,val x = true のように動作する。

ex3.3.2 **

1 設計方針

まず、複数の let 宣言を受け取るために syntax.ml にて構文を拡張する。

次に複数の let 宣言の入力を syntax.ml にて拡張した DeclList に変換するよう parser.mly に新たな文法規則を追加する。

そして、今までは eval.ml で 1 つずつ式を評価していたので複数の let 宣言を受け取り評価する関数を追加する。

最後に、cui.ml で複数宣言の結果を個別に表示するよう変更を加える。

2 実装にあたり工夫した点

parser.mly にて複数の let 宣言を 1 つのまとまりとして考えることとした。また、decl_list に let 宣言の (変数名, 式) を追加する際にメモリ効率を考え、中置演算子@を使わず:: で追加していき、toplevel で List.rev を使い逆順にして宣言の順番を合わせた。

また cui.ml にて List.fold_left を使うことで、parser.mly で合わせた順番通りに評価することで古い順で宣言の評価をできるようにしている。

3 動作確認

```
# let x = 1
  let y = x + 1;;
val x : int = 1
val y : int = 2
```

※型推論が行われているが、教科書の流れに従った変更の場合、val x = 1 val y = 2 のように動作する。

ex3.4.1(必修)

1 設計方針

MiniML3 インタプリタを作成するにあたって、fun 式による関数抽象と関数適用を追加するので BNF を拡張して fun 式と e e(関数適用) を追加する必要がある。

そのため、syntax.ml にて構文木の拡張を、lexer.mll にて予約語と記号の追加を、parser.mly にて新たに fun 式の追加と関数適用式が他の演算子より結合が強く左結合であることを踏まえた構文規則の追加を、行う。

また、eval.ml にて関数値をクロージャというデータ型で表現し、それを用いて eval_exp における FunExp と AppExp の評価方法を追加する。

2 インタプリタ動作確認

```
# let x = 3 in let f = fun y -> x + y in let x = 5 in f 8
val - = 11
```

fun 式による関数抽象、関数適用の機能を確認している。

ex3.4.3 *

1 設計方針

fun 式や let 式での複数引数を使う簡略記法をサポートするために、parser.mly にて複数引数を持つ関数定義式を補助関数を用いて右結合の入れ子関数に変換する。

つまり、構文上は `fun x y z -> e`,`let f x y z = e` のような形式を許しつつ、意味的には `fun x -> fun y -> fun z -> e`,`let f = fun x -> fun y -> fun z -> e` という形として考えている。

2 実装にあたり工夫した点

複数引数を持つ関数を 1 引数関数の入れ子として変換するという処理を挟むことで、MiniML3 の仕様から大きく変わらずに実現できた。また、複数引数を params という補助的な構文規則を用いることで変更を少なく抑えられた。

3 動作確認

```
# let add x y = x + y in add 2 3;;
val - = 5
```

ex3.5.1(必修)

1 設計方針

MiniML4 インタプリタを作成するにあたって、let rec 式、let rec 宣言を追加するので BNF を拡張して rec を追加する必要がある。

そのため、syntax.ml にて構文木の拡張を、lexer.mll にて予約語の追加を、parser.mly にて let rec 式と宣言の両方を追加するために toplevel での宣言と Expr における式の構文規則の追加そしてトークンの追加を、行う。

また、再帰的関数を in の後ろの式で用いる場合、関数自身は定義中で環境に含まれていないのでバックパッチで関数を環境の中に定義して後ろの式を評価するよう、eval.ml にて LetRecExp を追加すると同時にクロージャの環境を参照型に変更。

2 実装にあたり工夫した点

クロージャの環境を参照型に変更したので、FunExp や AppExp 内の環境を扱う部分を参照型に変更した。また、eval_decl におけるパターンマッチで RecDecl が新たに追加されたのでそこの補完を行った。その際、eval_exp での流れを利用しており最後の返り値のときに変数名、新しい環境、クロージャを返すようにしている。

3 インタプリタ動作確認

```
# let rec f = fun x -> x in f 10;;
val -= 10
# let rec fact = fun n -> if n = 0 then 1 else n*fact(n-1);;
val fact = <fun>
```

let rec 式、let rec 宣言の機能を確認している。

ex3.6.2 *

1 設計方針

リスト表記をサポートするので、BNF にリスト表記を追加する必要がある。

そのため、syntax.ml にて構文木を拡張し、リスト型と cons 演算子に対応した型を追加した。lexer.mll ではリストに使う記号と cons 演算子を追加。parser.mly では cons 演算子の結合の強さと結合性に注意しつつ、LTExpr と PExpr の間に CONSExpr を挿入し、AExpr の中に ListExp を用いてリストと空リストに対応するアクションを追加。

また、eval.ml では式が表す値として exval に ConsV,NilV を加え、eval_exp に ListExp と ConsExp の評価の仕方を追加する。

2 実装にあたり工夫した点

リスト表記に対応するにあたって、リスト表記 [1; 2; 3] や、:: を用いた cons 構文 1 :: [2; 3] の 2 つの入力の表現方法に対応するために構文木のデータ型を 2 つ増やし、評価する際は、ConsV にどちらも直すことで内部でのリスト表現を統一させた。

3 動作確認

```
# [3; 4; 5];;
val -= [3; 4; 5]
# 1 + 2 :: [2];;
val -= [3; 2]
```

感想

MiniML1 インタプリタから始まり、徐々に機能を追加していく工程を通して、各モジュールの働きやモジュール間の相互作用について多くの気づきがあった。また、新しい文法を追加する際の、BNF への追加→ syntax.ml での構文木のデータ型の追加→ lexer.mll での予約語、記号の追加→ parser.mly での構文規則と結合性、トークンの追加→ eval.ml での式の評価の追加、といった一連の流れがより身についたと思う。ほかにも、cui.ml にて REPL をループさせるためにほかのモジュールで一つずつ構文や評価をパスしていたのだなと実感した。