2018 年度 プログラミング D ML 総合演習課題

担当教員: 大倉 史生 < okura@am.sanken.osaka-u.ac.jp >

1　はじめに

総合演習では SML を用いた数式処理を行う．最終的には，前置記法で記述された（変数を含む）式と変数のマッピング表から，式の値を計算するプログラムを作成する．

前置記法とは，演算子を被演算子の前に置いて数式を記述する方法である．例えば，中置記法で `1 + 2` という数式を前置記法で記述すると `(+ 1 2)` となる．また，中置記法で `1 + 2 * 3` は前置記法で `(+ 1 (* 2 3))`，中置記法で `(1 + 2) * 3` は前置記法で `(* (+ 1 2) 3)` となる．読みやすさのために括弧を用いて記述したが，演算子が 2 項演算子のみであれば，前置記法では括弧を用いなくても曖昧さなく数式を表現することができる．

総合演習で作成するプログラムでは，前置記法で記述された`"(+ (* 10 a) (* b c))"`のような数式文字列と `[("a",1), ("b",2), ("c",3)]` のような変数マッピングリストから数式の値を計算する．この変数マッピングリストは `a=1`，`b=2`，`c=3` を表している．この例の場合，計算結果は `16` となる．

2　課題 1

課題 1 では，演算子を + だけに絞り，括弧も用いず，+ と非負整数のみからなる式を計算するプログラムを作成しよう．最初に，数式の文法を以下のように定義する．

<EXP> **::=** **非負整数** | "+"<EXP><EXP>

定義中の<EXP>が一つの式を表す．この記述は，<EXP>が，非負整数，または，"+"<EXP><EXP>の形をしているもの，のどちらかであることを表している．例えば，<EXP>は非負整数を表すため，1 や 2 は<EXP>で表される．よって，+ 1 2 も，"+"<EXP><EXP>の形になっているため，<EXP>で表される．さらに，+ + 1 2 3 も，+ 1 2 を一つの<EXP>とみなすことで"+"<EXP><EXP>の形となっており，<EXP>で表される．

これから上記の数式を処理するプログラムを考えていくが，数式文字列はそのままの形では扱いにくい．例えば，`"+ 10  20"`という数式文字列をそのまま文字リストに分解すると，

```
[#"+",#" ",#"1",#"0",#" ",#" ",#"2",#"0"]
```

となるが，スペースの部分は無視したいし，整数の部分は一続きで 10，20 のように扱いたい．そこで，上記の数式文字列を `["+","10","20"]` というような意味のある部分に区切る関数 `separate` を用意している．すなわち，関数 `separate` を用いることで，数式文字列を意味のある部分文字列に分解したリストを得ることができる．関数 `separate` の実行例は以下の通りである．

```
- separate "+ 1 1";
val it = ["+","1","1"] : string list
- separate "+ 10 * 20 30";
```

```
val it = ["+","10","*","20","30"] : string list
- separate "+10*20(30)";
val it = ["+","10","*","20","(","30",")"] : string list
```

また，separate で得られたリストの各要素は文字列であり，整数文字列を整数型の整数として扱うためには，文字列型から整数型へ変換する必要がある．この処理を行う関数 toInt も用意している．関数 toInt の実行例は以下の通りである．

```
- toInt "100";
val it = 100 : int
```

また，文字列が整数を表しているかどうかを判定する関数 isInt も用意している．その実行例は以下の通りである．

```
- isInt "100";
val it = true : bool
- isInt "+";
val it = false : bool
```

これらの関数は CLE 上の lib.sml ファイルの中で提供している．lib.sml を作業用フォルダにダウンロードし，

```
- use "lib.sml";
```

を実行すると，lib.sml 内の関数を利用できる．なお，関数 toInt，関数 isInt の引数は separate で分解された文字列を想定しており，"1a"のように数字とアルファベットが混ざった文字列に対しては正しく動作しない．このような文字列を含む数式に対しては，separate を適用した段階で例外 SyntaxError が発生する．

それでは，準備ができたので<EXP>で定義される式を計算するプログラムを考えよう．まず，処理の流れを考える．separate で分解された文字列リストが与えられれば，以下のように再帰的に前から順番に読み込みながら計算できる．

- 手続き EXP
  1. 文字列を 1 つ読み込む．読み込めなければ，エラーを発生させる．
  2. 読み込んだ文字列が非負整数であれば，その値を返す．
  3. 読み込んだ文字列が"+"であれば，以下の処理を行う．
     （a）再帰的に EXP を呼び出し，1 つ目の<EXP>に当たる値を得る．
     （b）再帰的に EXP を呼び出し，2 つ目の<EXP>に当たる値を得る．
     （c）2 つの値の和を返す．
  4. 読み込んだ文字列が非負整数でも"+"でもなければ，エラーを発生させる．

この手続きを SML 風に記述すると，以下のようになる．

```
01: fun EXP () =
02:   let
```

```
03:      val h = 文字列を 1 つ読み込んだもの．読み込めなければエラーを発生．
04:    in
05:      if isInt h then toInt h
06:      else if h = "+" then
07:        let
08:           val v1 = EXP ()
09:           val v2 = EXP ()
10:        in
11:           v1 + v2
12:        end
13:      else エラーを発生．
14:    end;
```

さて，これで残る問題は文字列をどのようにして 1 つずつ読み込んでいくかである．数式を標準入力やファイルなどのストリームから読み込んでいくのであれば，ストリーム用の関数を用いて上記のような形で関数を作成することができる（ストリームに関しては教科書 15 章を参照）．しかし，今回は引数として数式が与えられるため，ストリーム用の関数を用いて処理することができない．仕方がないので，関数 EXP の引数として数式を（関数 separate で分解した文字列リストの形で）与えることにしよう．このようにしておけば，1 つ目の文字列を読み込むのは，引数の文字列リストの先頭要素をとってくるだけで実現できる．この方針で関数 EXP を完成させられるか確認していこう．5〜6 行目の処理は先頭要素を使うだけなので問題ない．8 行目も，先頭要素を取り除いた残りのリストを与えて EXP を呼び出すことができるので問題ない．問題なのは，9 行目である．ここでも EXP を呼び出すのだが，この引数には 8 行目の v1 の計算で使った部分を取り除いたリストを与えなければならない．しかし，このままではリストのどこまでを v1 の計算で使ったかが分からないので，そのような引数を与えることができない．そこで，EXP で式の値を戻り値として返す際に，リストから使った部分を取り除いたものを一緒に返すようにしよう．EXP は式の値と残りのリストの 2 つを戻り値として返す必要があるが，これは 2 項組を返すことで実現できる．

上記の方針で作成した関数 EXP を以下に示す．

```
01: fun EXP nil = raise SyntaxError
02:   | EXP (h::t) =
03:       if isInt h then (toInt h, t)
04:       else if h = "+" then
05:           let
06:               val (v1,t1) = EXP t
07:               val (v2,t2) = EXP t1
08:           in
09:               (v1 + v2, t2)
10:           end
11:       else raise SyntaxError
```

関数 EXP の引数は，数式を関数 separate で分解した文字列リストとなる．例えば，以下のような出力を得

ることができる．

```
- EXP ["+","+","1","2","3"];
val it = (6,[]) : int * string list
- EXP ["+","1","2","+","3","4"];
val it = (3,["+","3","4"]) : int * string list
```

文字列リストを先頭から読み込んでいき，一つの<EXP>に該当する部分の式の値と残りの部分が 2 項組の戻り値として返ってきていることが分かるだろう．関数の中身は SML 風の関数とほとんど同じであるが，戻り値として式の値と残りの部分の組を返している．6 行目では，EXP の戻り値が組なので，val (v1,t1) = EXP t として，戻り値の 1 要素目と 2 要素目がそれぞれ v1 と t1 に格納されるようにしている．

では，上記の関数 EXP を用いて，数式文字列を引数として与えられる関数 compute を作成しよう．また，上記の EXP は式の後ろに余計なものがついている場合（"+ 1 2 +"など）にエラーを返せないので，その処理も追加しよう．最終的にできた関数は以下の通りである．

```
01: fun compute s =
02:   let
03:       fun EXP nil = raise SyntaxError
04:         | EXP (h::t) =
05:             if isInt h then (toInt h, t)
06:             else if h = "+" then
07:                 let
08:                     val (v1,t1) = EXP t
09:                     val (v2,t2) = EXP t1
10:                 in
11:                     (v1 + v2, t2)
12:                 end
13:             else raise SyntaxError
14:   in
15:       let
16:           val (result,rest) = EXP (separate s)
17:       in
18:           if rest = nil then result else raise SyntaxError
19:       end
20:   end;
```

プログラムは CLE 上の kadai1.sml に記述しているので，適当な引数を与えて実行してみよう．例えば，以下のような出力が得られるはずである．

```
- compute "+ + 1 2 3";
val it = 6 : int
- compute "+ + 100 20 + 1 2";
```

```
val it = 123 : int
```

3 課題 2

課題 2 では，以下の文法に対応するプログラムを作成する．

```
<EXP> ::= 非負整数 | <COMP>
<COMP> ::= "+"<EXP><EXP>
```

この文法は，課題 1 の文法の"+"<EXP><EXP>の部分を<COMP>を用いて表現しているだけである．そのため，表す数式は課題 1 の文法が表す数式と全く同じである．ここでは，<EXP>，<COMP>という 2 つの定義式を用いたときに，どのようなプログラムになるかを理解しよう．サンプルプログラムは以下の通りである．

```
01: fun compute s =
02:   let
03:       fun EXP nil = raise SyntaxError
04:         | EXP (h::t) =
05:             if isInt h then (toInt h, t)
06:             else if h = "+" then COMP (h::t)
07:             else raise SyntaxError
08:
09:       and COMP nil = raise SyntaxError
10:         | COMP (h::t) =
11:             if h = "+" then
12:                 let
13:                     val (v1,t1) = EXP t
14:                     val (v2,t2) = EXP t1
15:                 in
16:                     (v1 + v2, t2)
17:                 end
18:             else raise SyntaxError
19:   in
20:       let
21:           val (result,rest) = EXP (separate s)
22:       in
23:           if rest = nil then result else raise SyntaxError
24:       end
25:   end;
```

プログラムは CLE 上の kadai2.sml に記述しているので，適当な引数で実行させてみよう．課題 1 のプログラムから変化している部分は，3〜18 行目である．<EXP>，<COMP>に対応した 2 つの関数 EXP，COMP

が定義されている．この 2 つの関数は，EXP から COMP を呼び出し，COMP から EXP を呼び出すという相互再帰の関係となっている．そのため，and を用いて 2 つの関数を同時に定義している．関数 EXP の機能は課題 1 と同様である．関数 COMP の機能も関数 EXP と似ており，<COMP>に該当する部分の式の値を計算し，その残りの部分を組にして返す．

関数の動作は課題 1 とほとんど同じなので自分で確認してみよう．EXP では，非負整数の場合か<COMP>の場合があるが，<COMP>の場合と判断すると関数 COMP へ処理を丸投げしている．課題 1 で"+"<EXP><EXP>を処理した部分がほとんどそのまま関数 COMP の処理となっていることが分かるだろう．

4　課題 3

これからは課題 2 のプログラムを順に拡張していく．まず，課題 2 では演算子が"+"だけだったので，これを"−"，"*"，"/"にも対応させよう．この場合，数式は以下の形で定義される．

```
<EXP> ::= 非負整数 | <COMP>
<COMP> ::= "+"<EXP><EXP> | "−"<EXP><EXP> | "*"<EXP><EXP> | "/"<EXP><EXP>
```

では，この数式を処理するプログラムを作成してみよう．課題 2 のプログラムから関数 EXP，関数 COMP を以下のように変更すれば完成するはずである．

- 関数 EXP で，COMP に処理を丸投げする条件を変更．
- 関数 COMP に，h が"−"，"*"，"/"であった場合の処理を追加．整数型の割り算には，div を使おう．0 で割ると例外が発生するが，これに対応する必要はない．

5　課題 4

課題 3 までは括弧を用いずに数式を記述してきたが，読みやすさのために括弧を使うことにする．つまり，数式の定義を以下の形にする．

```
<EXP> ::= 非負整数 | <COMP>
<COMP> ::= "(""+"<EXP><EXP>")" | "(""−"<EXP><EXP>")"
         | "(""*"<EXP><EXP>")" | "(""/"<EXP><EXP>")"
```

この定義で表される数式は，"(+ 1 2)"や"(* (+ 10 20) (- 5 2))"という形となる．では，課題 3 のプログラムを変更して，この定義に対応する数式を計算するプログラムを作成してみよう．なお，括弧を無視すれば課題 3 と変わらないが，括弧の対応関係がとれていなければ（例えば，"(+ 1 2"などで）例外 SyntaxError を発生させるようにすること．

6　課題 5

次に，関数を使用できるようにプログラムを拡張しよう．ここでは，関数として，階乗を計算する fact とフィボナッチ数列の要素を返す fibo の 2 つを考える．そうすると，数式の定義は以下の形となる．

```
<EXP> ::= 非負整数 | <COMP> | <FUNC>
```

```
<COMP> ::= "(""+"<EXP><EXP>")" | "(""-"<EXP><EXP>")"
         | "(""*"<EXP><EXP>")" | "(""/"<EXP><EXP>")"
<FUNC> ::= "(""fact"<EXP>")" | "(""fibo"<EXP>")"
```

この定義で表される数式は，"(+ (fact (+ 2 3)) (fibo 5))"のような形となる．では，課題 4 のプログラムを変更して，この定義に対応する数式を計算するプログラムを作成してみよう．階乗，フィボナッチ数列を計算する SML の関数 fact，fibo は lib.sml の中で定義している．また，関数 EXP，COMP と同様に，関数 FUNC の定義が必要である．ここで，関数 EXP から関数 COMP または関数 FUNC へ処理を丸投げする必要があるが，<COMP>も<FUNC>も一文字目が"("なので，一文字目を調べるだけではどちらに丸投げすればよいかが分からない．そのような場合は，2 文字目も調べて，2 文字目が英字列であれば FUNC に丸投げし，そうでなければ COMP に丸投げするようにすればよい．文字列が英字列かどうかの判定には，lib.sml 内の関数 isAlp を用いればよい．

7 課題 6

最後の仕上げとして，課題 5 で作成したプログラムをもとに，変数に対応したプログラムを作成しよう．まず，数式で英字列を使用できるように，数式の定義を以下の通りとする．

```
<EXP> ::= 非負整数 | 英字列 | <COMP> | <FUNC>
<COMP> ::= "(""+"<EXP><EXP>")" | "(""-"<EXP><EXP>")"
         | "(""*"<EXP><EXP>")" | "(""/"<EXP><EXP>")"
<FUNC> ::= "(""fact"<EXP>")" | "(""fibo"<EXP>")"
```

ここでは，数式文字列とは別に変数マッピングリストを与える．変数マッピングリストは (string * int) list 型とする．リストの各要素は文字列と整数の組であり，例えば ("a",1) という要素は変数 a の値が 1 であることを意味する．では，数式文字列と変数マッピングリストを引数としてとり，数式の値を計算する関数 compute を作成しよう．compute の実行例は以下の通りである．

```
- compute "(+ (* 10 a) (* b c))" [("a",1),("b",2),("c",3)];
val it = 16 : int
- compute "(+ (* 10 a) (* b c))" [("a",10),("b",20),("c",30)];
val it = 700 : int
- compute "(fact (+ a b))" [("a",2),("b",3)];
val it = 120 : int
- compute "(+ abc def)" [("abc",10),("def",20)];
val it = 30 : int
```

ただし，数式で使われている変数が変数マッピングリストで定義されていない場合は，例外 NotDefined を発生させよ．

以下は，課題 6 のプログラムを作成するヒントである．これらのヒントには従っても従わなくてもよい．

- まず，変数マッピングリストと変数を表す文字列が与えられたとき，対応する値を返す関数を作成しよう．例えば，関数名を findValue とすると，実行例は以下のようになる．

```
- findValue "a" [("a",1),("b",2)];
val it = 1 : int
- findValue "b" [("a",1),("b",2)];
val it = 2 : int
```

この関数は以下のような形になるだろう．

```
fun findValue s nil = raise NotDefined
  | findValue s (h::t) =
      if [リストの最初の組 h が対象の s に該当する？]
          then [対応する値]
          else [findValue を再帰的に利用してリストの後続を検索];
```

関数を定義する際に unresolved flex record という例外が発生した場合は，引数の型が曖昧であるということなので，リストの各要素が `string * int` 型であることを明示すればよい．

- 作成する関数 compute は以下のような形になるだろう．

```
fun compute s mapL =
  let
      fun EXP nil = raise SyntaxError
        | EXP (h::t) = ...

      and COMP nil = raise SyntaxError
        | COMP (h::t) = ...
  in
      let
          val (result,rest) = EXP (separate s)
      in
          if rest = nil then result else raise SyntaxError
      end
  end;
```

関数 compute の引数として，変数マッピングリストを表す mapL を増やしている．数式の定義で課題 5 から変わっているのは<EXP>だけであり，関数 EXP に英字列に対応する処理を加えるだけでよい．関数 EXP は compute の中で局所的に定義されており，変数マッピングリスト mapL に直接アクセスすることができる．

8 拡張課題

時間に余裕のある者は，積極的に拡張課題にチャレンジしよう．拡張課題 1 および拡張課題 2 は課題 6 のプログラムからの拡張になっていますが，拡張課題 3 は独立した課題です．できる課題からやってもらってもかまいません．

8.1 拡張課題 1

拡張課題 1 では，課題 6 のプログラムを拡張して一つの演算子に対して 3 個以上の被演算子を書けるようにしよう．このように拡張しても，括弧を使っているため曖昧さは発生しない．これにより，"(+ 1 2 3 (* a b c))"のような数式を計算することができるようになる．ただし，減算，除算については，"(- 5 2 1)"が"5 - 2 - 1"として計算されるようにすること．数式の定義は以下のようになる．

```
<EXP> ::= 非負整数 | 英字列 | <COMP> | <FUNC>
<COMP> ::= "(""+"<EXP><EXP>{<EXP>}*")" | "(""-"<EXP><EXP>{<EXP>}*")"
         | "(""*"<EXP><EXP>{<EXP>}*")" | "(""/"<EXP><EXP>{<EXP>}*")"
<FUNC> ::= "(""fact"<EXP>")" | "(""fibo"<EXP>")"
```

ここで，{<EXP>}*は<EXP>を 0 個以上繰り返すことを意味している．

8.2 拡張課題 2

拡張課題 1 のプログラムをさらに拡張し，集合演算ができる計算機プログラムを作成しよう．まず集合を表す記号として”{” と”}” を導入する．例えば，{1 2 3} は 1, 2, 3 が要素である集合であり，{} は空集合を表す．集合演算としては和，積，および差の 3 つの演算を考え，それぞれの演算子として +，*，および−を導入する[*1]．これにより，``(+ {3 5 a} {b (* 2 3)})'' のような数式を計算できるようになる．

以上のような数式は，以下のように定義できる．

```
<SET>  ::= "{"{<EXP>}*"}" | <SCOMP>
<EXP> ::= 非負整数 | 英字列 | <COMP> | <FUNC>
<COMP> ::= "(""+"<EXP><EXP>{<EXP>}*")" | "(""-"<EXP><EXP>{<EXP>}*")"
         | "(""*"<EXP><EXP>{<EXP>}*")" | "(""/"<EXP><EXP>{<EXP>}*")"
<FUNC> ::= "(""fact"<EXP>")" | "(""fibo"<EXP>")"
<SCOMP> ::= "(""+"<SET><SET>{<SET>}*")" | "(""-"<SET><SET>{<SET>}*")"
         | "(""*"<SET><SET>{<SET>}*")"
```

なお，関数 compute は <EXP> を直接呼び出す形にしていたが，<SET> を呼び出すように変更する必要があることに注意すること．また，プログラム内部では集合をリスト型を用いて表現する．例えば，集合 {1 2 3} はプログラム内部では [1, 2, 3] とする．さらに，集合は以下の要件を満たすこと．

- 同じ要素は重複しない．例えば，[1, 2, 2] は [1, 2] と表現する．
- 集合内の要素の順番は気にしなくてよい．すなわち，[1, 2, 3] は [2, 3, 1] や [3, 2, 1] と表現しても良い．

拡張課題 2 のプログラムの実行例は以下の通りである．

```
- compute "(+ {1 2 a} {4 b c})" [("a", 3), ("b", 3), ("c", 5)];
```

[*1] 集合 A，B に対して，集合の和は $A + B := \{x | x \in A \vee x \in B\}$，集合の積は $A * B := \{x | x \in A \wedge x \in B\}$，集合の差は $A - B := \{x | x \in A \wedge x \notin B\}$ と定義する．

```
val it = [1, 2, 3, 4, 5] : int list
- compute "(* {1 2 a} {4 b c})" [("a", 3), ("b", 3), ("c", 5)];
val it = [3] : int list
- compute "(- {1 2 a} {4 b c})" [("a", 3), ("b", 3), ("c", 5)];
val it = [1, 2] : int list
```

8.3 拡張課題 3

中置記法の数式を計算するプログラムの作成にチャレンジしよう．これにより，`"(1+a)*(b+3*(c+2))"` のような自然な数式を計算することができるようになる．このような数式は，以下のように定義できる．

```
<EXP> ::= <TERM>{{"+"|"-"}<TERM>}*
<TERM> ::= <BASE>{{"*"|"/"}<BASE>}*
<BASE> ::= 非負整数 | 英字列 | "("<EXP>")" | <FUNC>
<FUNC> ::= "fact"<BASE> | "fibo"<BASE>
```

9 レポート

少なくとも以下の内容を含むレポートを作成せよ．

- 課題 3，課題 4，課題 5，課題 6 で作成したプログラムの説明（更新した部分だけでよい）．レポートの書き方については、Java 演習を参考に，プログラムを再現するのに十分な説明を行うこと．
- 拡張課題を行った場合は，作成したプログラムの説明．

作成したレポートを，ソースコードとともに以下の要領で CLE から **2 月 3 日 (日) 23:59** までに提出せよ．

- レポートとソースコードを一つの zip ファイルとしてまとめ，そのファイル名は `09Bxxxxx.zip` のように，学籍番号に拡張子を付与したものにすること．
- レポートは Word や LaTeX 等を用いて作成し pdf 形式で提出すること．なお，ファイル名は `09Bxxxxx.pdf` とすること．
- 課題 6 および，（行っていれば）拡張課題 1〜3 まで作成したプログラムを，それぞれ別ファイルとしてまとめること．その際，ファイル名は `09Bxxxxx.sml`（課題 6），`09Bxxxxx-1.sml`（拡張課題 1），`09Bxxxxx-2.sml`（拡張課題 2），`09Bxxxxx-3.sml`（拡張課題 3）とすること．`lib.sml` 以外で作成した関数は必要とするファイルにまとめておくこと．すなわち，`09Bxxxxx.sml` の中に `use ``lib.sml'';` は含めずに，`lib.sml` と `09Bxxxxx.sml` のみを読み込んで課題 6 のプログラムを実行できるようにしておくこと．
- 例外 `SyntaxError`，`NotDefined` は `lib.sml` の中で定義している．動作確認の都合上，この 2 つの例外を `lib.sml` 以外で新たに定義しないようにすること．