プログラミングD SML レポート

担当教員　小南大智
提出日　2024 年 2 月 6 日
氏名　山久保孝亮
学籍番号　09B22084
メールアドレス　u327468b@ecs.osaka-u.ac.jp

1　課題 3 のプログラムの説明

課題 3 では COMP の計算の処理を足し算のほかに引き算, 掛け算, 割り算を追加した.

1.1　変更点

変更点は以下の通りである.

- 関数 EXP から関数 COMP に移動する条件の変更.
- それぞれの演算の処理の実装.

1.2　処理の実装

変更点で述べた内容の詳細および実装法について述べる.

1. 関数 EXP から関数 COMP に移動する条件を以下のように変更した.

Listing 1: 処理の実装

```
1         if h = "+" orelse h = "-" orelse h = "*" orelse h = "/"
```

2. 足し算の演算の処理と基本的な構造はすべて同じであるので代表して足し算の処理を記述する.EXP 関数を残りの文字列 t に対して呼び出して数値を v1 に一つ取得する. そのとき t から v1 を除いた残りを t1 とし,t1 に対してもさらに EXP を呼び出し, もう一つ数値 v2 を取得する. そして t1 から v2 を除いた文字列と v1 と v2 を足し合わせたものを COMP 関数の返り値として返す.

Listing 2: 処理の実装

```
1         if h = "+" then
2                 let
3                     val (v1,t1) = EXP t
4                     val (v2,t2) = EXP t1
5                 in
6                     (v1 + v2, t2)
7                 end
```

1.3　テスト結果

以下の図 1 ように加減乗除 4 つの四則計算を行えるようになった.

2　課題 4 のプログラムの説明

課題 4 では括弧を使って数式を計算できるようにした。以下でその変更点について述べる.

2.1　変更点

変更点は以下の通りである.

1. EXP 関数で”(”を取得できるようにした.
2. COMP 関数で”(”と対応する”)”内の処理を実行して二つの括弧を計算式の文字列から削除する.

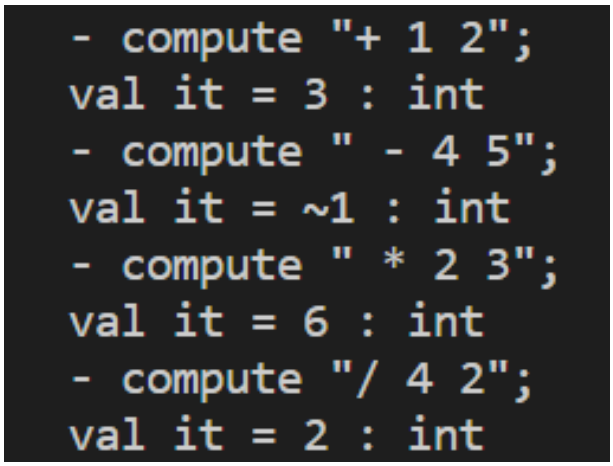

図 1: 四則演算のテスト結果

2.2 変更部分のコード

Listing 3: EXP 関数の一部

```
fun EXP nil = raise SyntaxError
| EXP (h::t) =
  if isInt h then (toInt h, t)
  else if h = "(" orelse h = "+" orelse h = "-" orelse h = "*" orelse h = "/"
      then COMP (h::t)
  else raise SyntaxError
```

Listing 4: COMP に新たに追加した分岐

```
if h = "(" then
let
    val (v1,t1) = EXP t

    fun splitList lst =
        case lst of
            [] => raise SyntaxError
        | hd::tl => (hd, tl)

    val (head,t2) = splitList t1
in
    if head = ")" then
        let
        in
            (v1, t2)
        end
    else
        raise SyntaxError
end
```

2.3　処理の実装

変更点で述べた内容の詳細および実装法について述べる.

1. 括弧を用いた数式の計算を実装するために Listing 3 の 4 行目のように EXP 関数内の演算子の判定に”(”を追加した. これにより,EXP 関数で与えられた文字列の最初の文字が”(”であるときも COMP 関数に飛ぶことができるようになった. また, 演算子と同等の扱いにした理由は”(”の中に再び”(”があった際に再帰的に呼び出すことができるようにするためである.

2. Listing 4 の 1 行目のように COMP 関数内の条件分岐に先頭の文字が”(”である場合を追加した. この分岐内では 3 行目のように一度 EXP 関数が呼び出されて v1 に先頭の文字が,t1 に残りの文字列が格納される. このとき,v1 が”(”であった場合と演算子であった場合に処理が分岐する.

   - 先頭が演算子の時
     COMP 関数が呼び出されて課題 3 で実装した計算の処理が実行される. 計算された後に v1 に入るのは計算結果の数値で,t1 に入るのは元の文字列から”(”と, 計算に使用した文字と, 数字 2 つを取り除いた文字列が格納される.
   - 先頭が”(”のとき
     再び COMP 関数が呼び出されてその中の”(”の条件分岐内で EXP 関数が同じように呼び出される.

   その後, 残りの文字列の先頭を確かめるために 5 から 8 行目のように関数 splitList を定義している. そして 10 行目で splitList を使用して t1 を head と t2 に分解している. 正しく数式が入力されていれば head には”)”が入力されているはずであるので 12 行目のように判定して一致していれば v1 と t2 を返し, 一致してなければ SyntaxError を返す. 再帰的に呼び出された際も”(”と”)”をセットで処理を行うので対応関係は守られる.

2.4　テスト結果

以下の図 2 の一つ目のように括弧を含めた数式の計算ができるようになり, 二つ目の実行例のように括弧の対応関係が取れていない数式に対しては SyntaxError を返すことができるようになった.

```
- compute "(+ 1 (* 2 3))";
val it = 7 : int
- compute "(+ 1 (* 2 3)";

uncaught exception SyntaxError
  raised at: kadai4.sml:17.42-17.53
```

図 2: 課題 4 のテスト結果

3　課題 5 のプログラムの説明

課題 5 では”fibo”と”fact”という計算するための関数を使えるようにした.

3.1 変更点

変更点は以下の通りである.

1. 新たに関数 FUNC を定義し, その中に”括弧の処理と fact”と”fibo”の処理を記述した.
2. EXP 関数で先頭の文字だけでなく二文字目も取り出して”(”の次の文字も判定するようにした.

3.2 処理の実装

変更点で述べた内容の詳細および実装法について述べる.

1. 先頭の文字が”(”か”fact”か”fibo”で条件を分岐させる.”(”内の処理は課題 4 のときの処理と同じである. また, これまでとは違い, 数式の文字列を先頭の文字である h と先頭から二番目の文字である h1 とそれら以外残りの文字列である t に分けている.

Listing 5: EXP 関数の一部

```
1        if isInt h then (toInt h, [h1]@t)
2            else if h = "(" then
3                if h1 = "+" orelse h1 = "-" orelse h1 = "*" orelse h1 = "/" then
                     COMP (h::[h1]@t)
4                else if isAlp h1 then FUNC (h::[h1]@t)
5                else raise SyntaxError
6            else if h = "+" orelse h = "-" orelse h = "*" orelse h = "/" then
                 COMP (h::[h1]@t)
7            else if isAlp h then FUNC (h::[h1]@t)
```

また, ここで”(”の条件分岐を考えている理由としては, 上の Listing5 の EXP 関数の FUNC に飛ぶ際の処理で h が”(”と一致すればさらに次の要素である h1 がアルファベットであったときに”(”から始まる文字列を渡しているためである. ここでは h1 と t を連結した文字列がそれに対応する.”(”から渡していないと, その”(”に対応する”)”が出てきたときにエラーが起きてしまうためである. また,”fibo”と”fact”は lib.sml で実装されていたのでそのまま使用した.

2. これまでは”(”と演算子の分岐を同時に行っていたが, 新たな関数 FUNC に分岐させるために”(”はほかの演算子たちとの分岐とは独立させ, またその中にさらに次の要素が何であるかを判定する分岐を作成した. 具体的には次の要素が演算子であるかアルファベットであるかという条件分岐を作成した. 演算子であれば COMP を呼び出し, アルファベットであれば FUNC を呼び出して処理を行う.

3.3 テスト結果

以下の図 3 のように fibo や fact を含めた数式の計算を実行できるようになった.

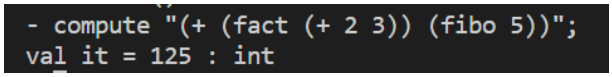

図 3: テスト結果

4　課題 6 のプログラムの説明

課題 6 では数式文字列以外の引数として変数マッピングリストをとるようにし, 数式中に登場した変数名に対応する数値をそのリストから取り出して置き換えることができるようにした.

4.1　変更点

具体的なコードの変更点は以下の通りである.

1. 関数 compute に二つ目の引数 mapL の追加.
2. 注目している文字がマッピングリストに存在すれば対応する数値を取り出して返す関数 findValue の実装.

4.2　処理の実装

変更点で述べた内容の詳細および実装法について述べる.

1. 関数 compute が文字列 s だけでなく (string * int) 型の要素を持つリストである mapL も引数に持つので, このとき型明示は変更点のコードのように行った.

Listing 6: 型変換

```
1    fun compute (s : string) (mapL : (string * int)list) =
```

また, マッピングリスト内の変数名と課題 5 で実装した”fact”と”fibo”を区別するために先に先頭の文字が”fibo”と”fact”のいずれかであるかを判定してその後にアルファベットであるかどうかの判定を行った.

Listing 7: EXP 関数の一部

```
1        fun EXP nil = raise SyntaxError
2          | EXP (h::h1::t) =
3            if isInt h then (toInt h, [h1]@t)
4            else if h = "(" then
5                if h1 = "+" orelse h1 = "-" orelse h1 = "*" orelse h1 = "/" then
                     COMP (h::[h1]@t)
6                else if isAlp h1 then FUNC (h::[h1]@t)
7                else raise SyntaxError
8            else if h = "+" orelse h = "-" orelse h = "*" orelse h = "/" then
                 COMP (h::[h1]@t)
9            else if h = "fact" orelse h = "fibo" then FUNC (h::[h1]@t)
10           else if isAlp h then (findValue h mapL,[h1]@t)
11           else raise SyntaxError
```

先頭の文字が”fibo”と”fact”のいずれかであった場合は FUNC を呼び出して課題 5 で記述した通りの処理を行う. 先頭の文字がアルファベットであった場合は findValue 関数を呼び出す. このときの引数は先頭のアルファベットと mapL である.findValue 内の具体的な処理は 2 で記述する.

2. findValue 関数では引数のアルファベットが mapL の中に存在するかを findValue 自身を再帰的に呼び出すことで判定している. リストを先頭の要素である s とそれ以外の要素である t に分解し,s の一つ目の要素を name, 二つ目の要素を value と型明示している. name が引数として与えられた存在する

か判定したいアルファベットである s と一致しているかどうかを判定し, 一致していれば value を返す. 一致していなければ s とリストの残りの要素 t を findValue 関数に渡し同じように処理を行う. 最後の要素まで確認して name と一致しなければ SyntaxError を返す.

4.3 テスト結果

以下の図 4 のように数式中にマッピングリスト内に登場する変数を利用できるようになった.

```
-  compute "(+ (* 10 a) (* b c))" [("a",1),("b",2),("c",3)];
val it = 16 : int
```

図 4: テスト結果

5 拡張機能の説明

今回私は拡張機能の 1 番に取り組んだ.

5.1 変更点

COMP 関数の四則演算の処理を変更した.

5.2 処理の実装

引き算と割り算のときに計算結果が合わなくなってしまうため, 前の二つから計算を実行するようにして実現した. 途中の計算の種類が変わるだけなので足し算の時の処理のみを代表的に説明する. 具体的には COMP 関数でどの演算子なのかを取り出してから 2 つ先までの数値を EXP 関数を 2 回呼び出して取り出す. そしてそれらを足し合わせて v という変数に格納しておく. そのあと課題 4 の時と同様に splitList 関数を定義して 3 番目の要素を文字列として取り出す. その 3 番目の要素が数値でなければ v が返り値となる. そして数値であれば spllitList 関数で先頭から 4 番目の文字を取り出して 3 番目の要素を数値に変換する.4 番目の要素が数値でなければ v に 3 番目の要素を足したものが返り値となり, 数値であれば演算子,v の値,3 番目の要素を含めたそれ以降の文字列を結合し COMP を再帰的に呼び出す.

5.3 テスト結果

以下の図のように数式中に数値だけが登場するのであれば正しく計算することができた.

```
- compute "(+ 1 2 3)"[];
val it = 6 : int
```

図 5: 拡張課題のテスト結果

6　今回の課題に対する考察

拡張課題 1 において変数が数式の最後の方に登場してしまうとエラーが発生してしまった. これの原因としては, ”+ 1 2 3”のように連続した計算で数式が終了する場合の処理で 4 つ先まで EXP 関数を使わずに読んでしまうため findValue が呼び出されない事が考えられる. ただ, この 4 つ先まで読むという処理を実装した理由としては compute 関数において数式を先頭, 二番目, 残りの三つに分けたことで残りの部分に何も入らなくなるとエラーが発生してしまったからである. なので数式を先頭と残りに分け, 課題 4 の if 文の let 環境内で次の文字を splitList 関数を使って実装