知能プログラミング演習 II 課題 2

グループ 8
29114060 後藤 拓也

2019 年 10 月 27 日

■提出物　rep2

■グループ　グループ 8

■グループメンバー

学生番号	氏名	貢献度比率
29114003	青山周平	no
29114060	後藤拓也	no
29114116	増田大輝	no
29114142	湯浅範子	no
29119016	小中祐希	no

■自分の役割　必須課題 2.1.(2)
「複数のパターンが与えられたときに全ての可能な変数束縛の集合を返すようなプログラムの作成」

1　課題の説明

複数のパターンが与えられたときに全ての可能な変数束縛の集合を返すようなプログラムを作成せよ．例えば，上記の例で「?x is a boy」と「?x loves ?y」の両方が与えられたときに，(?x, ?y) の全ての可能な変数束縛の集合として (Taro, Jiro), (Jiro, Hanako) を返すこと．

2　手法

課題を達成する (つまり,「?x is a boy」と「?x loves ?y」と引数を取ると, (Taro, Jiro), (Jiro, Hanako) を返す) ために以下のようなプログラムの処理を行う.

1. 全体をまとめるハッシュマップを 1 つ生成する.
2. 1 つ目の引数, 2 つ目の引数を順番に処理を行う.
3. text ファイルからデータセットを 1 行ずつ読み取り, 1 行ずつ引数と照合する.
4. 1 行はトークン単位に分割され, マッチングは各トークンごとで行う.
5. 対応した場合, ハッシュマップに Key として変数 (?x や?y など) を, Value として具体定数 (Taro や Jiro など) を加えていく.
6. 1 つ目の引数で束縛された変数は, 2 つ目の引数に反映される (1 つ目で?x=Taro と決まれば, 2 つ目の引数ではその条件を引き継がせる).
7. ハッシュマップを 2 次元リストにし, 中身 (Key や各 Key に対応する Vlaue) をリスト化することで, 全てのマッチング可能なパターンを 1 つのハッシュマップに格納できる.

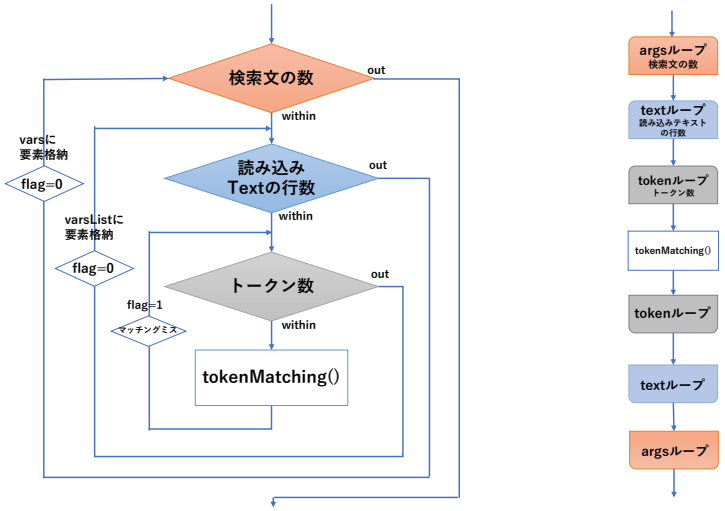

図 1　unify プログラムの 3 つのループ

プログラムの流れは多段階のループ構造を構築している. 図 1 における”3 つの大きなループ”を参照し, 上記の 7 つの手法について詳しく説明する.

1. に関して, main クラスで 1 度 Unifier クラスのインスタンスが生成されるとコンストラクタにより, ハッシュマップをはじめとするあらかじめ宣言されていた複数の変数やリストを初期化する. ここで初期化され生成されたインスタンスは, 今後のプログラム全体で使われる.

2. に関しては, 2 つの引数をリストに格納させ, ループ処理を行う. ここが 1 つ目の大きなループである.

3. に関しては, BufferedReader オブジェクトによりテキストを 1 行ずつ読み込む. ここが 2 つ目の大きなループである.

4. に関しては, 既存のプログラムを利用し, StringTokenizer により分解したトークンの数だけループ処理をする. ここが 3 つ目の大きなループである. 引数と読みこんだ 1 行とでトークン数の異なる場合は, トークン数の多いほうにそろえるように, 少ないほうに null を入れて数を調整する.

5. に関しては, トークン単位のマッチングは成功しても, 1 行を通して成功していなければ成功とはいえない. 例えば, 「?x is a boy」と「Hanako is girl」は, トークン単位のループ処理ででマッチングをしていくと, 先に「?x = Hanako」でマッチングが成功するが, その後「boy != girl」となるので, 今回は失敗であり, ハッシュマップには追加できない. これを実現するために, 1 行すべてのマッチングが成功した後に, これまで成功していたトークンの組をハッシュマップに保存する処理を行う. 具体的には, マッチングが成功したトークンの組はそれぞれ一時バッファならぬ一時リストに追加して, ハッシュマップには追加しない. マッチングの成功不成功を判断する”ミスフラグ”を用意し, 1 行におけるトークンの組み合わせすべてが成功したら (ミスフラグが上がらなかったら), 先ほどの一時リストの要素をハッシュマップに保存する.

6. に関しては, 「?x = Taro, Jiro」と 1 つの変数において, 2 つの定数が束縛されることがあるので, それぞれにおいて 2 つ目の引数を変更する. 具体的には, 2 つ目の引数は「?x loves ?y」から「Taro loves ?y」と「Jiro loves ?y」の 2 つに変更させる. そのため, 手法 1 説明したで 1 つ目の大きなループは, 引数 2 つであっても, 3 回ループが行われる.

7. に関しては, 上記の手法 6 で述べたように, 1 つの Key(たとえば?x) に対して, 複数の Value 値 Taro, Jiro を持つので, リスト化する必要があった. 上記の手法 6 で”トークン (語) の単位で行うのではなく, 1 行単位でハッシュマップに保存をする”と述べたが, この際, 実際のマップに保存するわけではなく, 2 次元リストであるハッシュマップに格納するためのリストに保存している.

3 実装

上記の 7 つの手法の中で, 特に重点を置いた箇所に関して, プログラムを参照にしつつ説明していく.

手法 5 に関する「1 行を通してすべてのトークンがマッチングに成功した時のみ, ハッシュマップに反映させる」の部分を実装したソースコード 1 に示す。

ソースコード 1　1 文すべて終わったら格納する

```
1
2   for(int i = 0 ; i < length ; i++){ //1語ずつマッチングしていきます
3           System.out.println("flag = " + flag);
4
5           if(!tokenMatching(buffer1[i],buffer2[i])){ //マッチングできてないなら...
6                   System.out.println("Search Error 2");
7
8                   flag = 1; //miss フラグ上げる
9           }
10
11  } //1文の解析がすべて終わって...
12  if(flag == 0) { //missflag が一度も上がっていないなら,
13          for(int len = 0; len < ValueList.size(); len++) {
14              //HashMap に格納せずに,
15              //vars.put(KeyList.get(len), ValueList.get(len));
16              System.out.println("ValueList.get("+ len + ") ="+ValueList.get(len));
17
18              //varslist に格納
19              varslist.add(ValueList.get(len));
20          }
21  } //Text の文全てが終わったら...
```

flag はフィールド変数として, Unifier クラスのどのメソッドでも用いられる. 上記には書かれていないが, tokenMatching メソッドの varMatching メソッド内で 1 単語 (トークン) をマッチングさせるたびに, マッチング成功の成否に合わせ flag 管理をしている.

また, ハッシュマップは実際には 2 次元リストにより構築されているため, 1 文解析がすべて終わり, フラグが立ってない場合の処理だが, 手法 5 の説明時には簡略化のために省略したが, 実際には"ハッシュマップに格納する"のではなく, "ハッシュマップに格納するためのリスト varslist に登録する"である.

手法 6 に関する「1 つ目の引数により得られた束縛条件を 2 つ目の引数に反映する」の部分を実装したソースコード 2 に示す。

ソースコード 2　第 1 引数から第 2 引数への変数束縛条件の引継ぎ

```
1  //1つ目の引数,2つ目の引数,順番に処理する！
2  for(int num = 0; num < args.size(); num++) {
3
4      if(num == 1) { //2回目で
5          //変数 (?x,?y などなど)の数だけ...
6          for(int keyNum = 0; keyNum < KeyList.size(); keyNum++) {
7              //その変数に対応する値 (Taro, Jiro などなど)の数だけ
8              for(int valueNum = 0; valueNum < varslist.size(); valueNum++) {
9
10                 //文字列の中に?が入っていたら...
11                 if(string2.contains("?")) {
12                     //文字列として置き換えて,新しく作成！
13                     String stringx = string2.replace(KeyList.get(0), varslist.get(
                           valueNum));
14                     System.out.println("stringx = " + stringx);
15                     args.add(stringx); //などなど
16                 }
17             }
18         }
19         args.remove(1); //もとを置き換え
20         System.out.println("args.size() = " + args.size());
21     }
22
23     varslist = new ArrayList<String>(); //複数人対応のValue:値を格納させる.
```

1 つ目のマッチングの結果で得られた Key を格納した KeyList, 各 Key に対応した Value を格納した varslist が用いられている. 具体的には, 1 つ目の引数で"?x is a boy"としたとき, ?x = Taro, Jiro が当てはまるため, KeyList には「?x」が, varslist には「Taro, Jiro」が含まれている. これを replace メソッドを用いて, 実際に「?x を Taro」という風におきかえている.

ポイントとしては, 引数は 2 つでも, 1 つ目の引数のマッチングの結果, 「?x = Taro, Jiro」のように複数対応する場合, 2 つ目の引数が「?x loves ?y」の際, ?x が 2 つ代入され, 「Taro loves ?y」と「Jiro loves ?y」と増えることである. そのため, 元あった「?x loves ?y」を消すために. remove メソッドで代入前の 1 文を消している.

手法 7 に関する「ハッシュマップの 2 次元リスト化による要素代入」の部分を実装したソースコード 3 に示す。

ソースコード 3　ハッシュマップの 2 次元リスト化

```
1   if(flag == 0) {
2
3       //いま見てるKey の番号にしないと！
4         int index = KeyList.indexOf(String.valueOf(keyList.get(0)));
5         System.out.println("KeyList = " + KeyList.get(index));
6
7         /* 参照渡しだから,作ったリストを削除してももとの場所をささない.*/
8         ArrayList<String> array = new ArrayList<>();
9         if(vars.containsKey(KeyList.get(index))) { //すでに作ったことがあったら,
10            System.out.println("削除します");
11            array.clear(); //消す
12            array = arrayCopy;//コピーを戻す
13                  flag2 = 1;
14        }
15        arrayCopy = array; //コピーを取っておいて,
16        array.add(ValueList.get(0));
17        if(flag2 == 0)
18             vars2list.add(array);
19        System.out.println("vars2list() = " + vars2list.toString());
20        vars.put(KeyList.get(index), vars2list.get(index)); //改良HashMap に格納
21
22        flag2 = 0;
23  }
24   System.out.println("途中結果は" + vars.toString() + "\n");
25   flag = 0; //falg のリセット
```

ここの処理は Text の全ての行が読み込めたら行われる. 2 次元リストを作るためには, リストに格納する処理を行うが, その際, Key の数 (?x だけなら 1 つ, ?x と?y なら 2 つなど）に応じて, 複製する Value のリストの数も変わってくる. リストの名前には, 配列の buffer[i] のように数字を名前に付け加えることができないため, ループ処理を行うこのプログラム中では, とりあえず, リストを作成するが, すでにある Key に対してリストが存在していれば, ダブってしまうので, 削除する. このとき, java ではリストのオブジェクトも参照渡しであるので, ただ消すだけでは, 参照先を前のリストに戻すことができない. そのために, 前のリストをコピーしておき, 消した際には, コピーもとを参照させる.

ソースコード 4　複数候補に対応した varMatching メソッド

```
1  boolean varMatching(String vartoken,String token){
2    /* (注意)
3      * すでに?x=Taro という制約がある状態で,さらに?x=Jiro を付け加えないといけない.
             うやむやにやっても,かぶっている候補を増やしてしまうだけである. */
4   //すでにあるKey に対して...
5      //HashMap に vartoken というキー(Not 値)が存在するかどうか
6      if(vars.containsKey(vartoken)){
7              System.out.println("varslist.size() = " + varslist.size());
8
9              //まだvarslist には入ってないけど,Maticng 成功した場合,
10             if(varslist.size() == 0) {
11                     ValueList.add(token);
12                     keyList.add(vartoken);
13             }
14             //普段はこっちだよね
15             else {
16                     for(int i = 0; i < varslist.size(); i++) {
17                         //すでに登録されている関係なら...
18                         if(token.equals((vars.get(vartoken)).get(i)))
19                                 System.out.println("格納済み");
20                         else { //別の値 (Taro じゃなくて Jiro)が来たら, 加えます.
21                                 ValueList.add(token);
22                                 keyList.add(vartoken); //Key も含めて再登録しないと...
23                         }
24                     }
25             }
26             return true;
27     }
28
29   //初めてのKey に関して...
30     else {
31             replaceBuffer(vartoken,token);
32             //HashMap の値リスト:varslist に含まれているかどうかで見る
33             if(varslist.contains(vartoken)) {
34                 replaceBindings(vartoken,token);
35             }
36             //vars.put(vartoken,token); //ここではハッシュマップに登録しません.
37             if(!KeyList.contains(vartoken)) { //1回だけでっせ！
38                     KeyList.add(vartoken);
39             }
40             ValueList.add(token);
41             keyList.add(vartoken);
42     }
43     return true;
44 }
```

上記は, 実際に 1(トークン) 単語単位でマッチングを行う varMatching メソッドである. ValueList と keyList は, テキストの 1 行ずつで初期化されるようになっており, ここのメソッド内でしかこの 2 つのリストは操作されない. この 1 行単位で初期化されるリストの結果は, 全体に反映されるリスト varslist に別のメソッドで格納されるので, それをもとに if 文の条件分岐を行っている.

4 実行例

Fibonacci クラスに引数 5 を指定した実行結果を以下に示す。

```
1 clf1XXXX@cse:~/eclipse−workspace/Fibo > java Fibonacci 5
2 8
```

今回は 0, 1 番目のフィボナッチ数を 1 としたため、calcFibo() の引数に 5 を与えた時は 8 となるのが正しい動作である。これは、・・・

5 考察

6 に関して, 1 つ目の引数で束縛された変数の値を, 実際に 2 つ目の引数に代入するので, もし「?x loves ?y」と「?x is a boy」という順番の引数の組み合わせだと最終的に間違ったハッシュマップが生じる. というのも, 「?x loves ?y」にあたる?x は [Hanako, Taro, Jiro] であるので, ハッシュマップその組み合わせが登録されるが, 「Hanako is a girl」なので, ミスが生じるが既にハッシュマップに登録されているので間違った出力となる. これは unify していない. ただの Matching である.... 引数も 2 つに限定させてしまった.

6 感想

まじで難しかった. 変数の大文字小文字がかなりザツでプログラム作成の後半になればなるほどわかりにくかった.

参考文献

[1] Java による知能プログラミング入門 –著：新谷 虎松

[2] java CSV 出力 –著：TECH Pin
https://tech.pjin.jp/blog/2017/10/17/【java】csv 出力のサンプルコード