CS536 P1 part1 description – Yusen Liu

Below is the description of P1.java. Note that, only the print() will produce output. Besides that, P1.java produces output only if a test fails.

1. The first thing my program did was to call the constructor of Sym class. I initialized an ArrayList of length 10 (lenSym), named symList, with the first 5 Sym's of type "int", and the last 5 Sym's of type "char", which is shown below:

```
// initialize an arraylist of Sym of length 10, with the first five of
// type "int", and the last five of type "char"
ArrayList<Sym> symList = new ArrayList<Sym>();

for(int i = 0; i < lenSym/2; i++){
    symList.add(new Sym("int"));
}
for(int i = 0; i < lenSym/2; i++){
    symList.add(new Sym("char"));
}
```

2. In order to test the functionality of getType(), I checked if the 0^{th} element in symList has type "int". If not, an error message will output to terminal. The code is shown below:

```
// test of getType()
// the 0th element should have type "int"
if(!symList.get(0).getType().equals("int")){
    System.out.println("getType() failed");
}
```

3. Now that getType() works well, we can test if the constructor works well using getType(). I used getType() to test if the 0^{th} and 4^{th} elements of symList had the same type. By my initialization, they should have the same type. If they have different types, then error message will output to the terminal, meaning that the constructor did not work well. The code is as follows:

```
// test of constructor
// the 0th and 4th element should have the same type
if(symList.get(0).getType().equals(
        symList.get(lenSym/2-1).getType()) == false){
    System.out.println("constructor in Sym failed");
}
```

4. In order to test the functionality of toString(), I took the similar approach, but compared the 4^{th} and 9^{th} element to see if they have different type. By my initialization, they should have different types. If they have the same type, then an error message will output to the terminal. The code is as follows:

```
// test of toString()
// the 9th and 4th element should have different types
if(symList.get(lenSym - 1).toString().equals(
        symList.get(lenSym/2 - 1).toString()) == true){
    System.out.println("toString() in Sym failed");
}
```

5. Since in P1 part1, both getType() and toString() has the same functionality, I conducted the following test to compare if they would return the same type. Note that this test will only work in P1 part1.

```
// combination test of getType() and toString()
// note that this test ONLY works in P1a
if(symList.get(lenSym/2 + 1).getType().equals(
        symList.get(lenSym/2 + 1).toString()) == false){
    System.out.println(
        "getType() and toString in Sym are not the same in P1a");
}
```

6. Next, we could start to test the functionalities of methods in SymTable class. I first constructed a SymTable named symTable using the constructor of SymTable class. Since it is stated that the constructor should initialize the SymTable's List field to contain a single, empty HashMap, we could call print() to test if the constructor had initialized an empty HashMap. The code is as follows:

```
// test of constructor
// symTable should consist of one empty hashmap
SymTable symTable = new SymTable();

// test of print() and check if constructor has initialized an empty
// HashMap
symTable.print();
```

We will see the first output as follows:

```
Sym Table
{}
```

7. Next, we could use lookupLocal() to test the functionality of addDecl(). Note that, we assume lookupLocal works well. The remaining functionalities of lookupLocal() cannot be tested until we have added at least two HashMaps into the SymTable (whose name is symTable). What I did was to add 3 name-Sym pairs to the first HashMap using addDecl(), and then use lookupLocal() to check if it could return the Syms according to the names. The code is as follows:

```
// use lookupLocal() to test addDecl()
try{
    // add the given name and sym to the first HashMap in the list
    symTable.addDecl("name11", symList.get(0)); // int
    symTable.addDecl("name12", symList.get(7)); // char
    symTable.addDecl("nameSame", symList.get(6)); // char

    // use lookupLocal() to test if addDecl() worked well
    // if any of the names cannot be found, then addDecl() failed
    if(symTable.lookupLocal("name11") == null
            && symTable.lookupLocal("name12") == null
            && symTable.lookupLocal("nameSame") == null){
        System.out.println("addDecl() failed to add names");
    }
}
catch(DuplicateSymException ex){
    System.out.println("DuplicateSymException occurs");
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

8. After step 7, I called addScope() to add a new HashMap, and added another three name-Sym pairs. This step is similar to step 7. Note that one of the three names, “nameSame”, is the same as the name “nameSame” in step 7. The code is as follows:

```
// test of addScope()
// add a new HashMap to the first
symTable.addScope();

try{
    // test of addDecl()
    // add the given name and sym to the first HashMap in the list
    symTable.addDecl("name21", symList.get(7)); // char
    symTable.addDecl("name22", symList.get(0)); // int
    symTable.addDecl("nameSame", symList.get(0)); // int

    // use lookupLocal() to test if addDecl() worked well
    if(symTable.lookupLocal("name21") == null
            && symTable.lookupLocal("name22") == null
            && symTable.lookupLocal("nameSame") == null){
        System.out.println("addDecl() failed to add names");
    }
}
catch(DuplicateSymException ex){
    System.out.println("DuplicateSymException occurs");
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

9. Now, it’s a good time to test the functionality of print(), and to check what the symTable is like after we have added the second HashMap. The code is as follows:

```
// test functionality of print()
symTable.print();
```

The output will be like:

```
Sym Table
{name22=int, name21=char, nameSame=int}
{name12=char, name11=int, nameSame=char}
```

10. Now, we can test the functionality of lookupLocal().
    - If we call lookupLocal(“nameSame”), it should return the Sym of type “int” (in the first HashMap in symTable) instead of “char” (in the second HashMap in symTable).
    - If we call lookupLocal(“name21”), it should return the Sym of type “char”.
    - If we call lookupLocal(“nameNotExist”), it should return null.
    - If we call lookupLocal(“name11”), it should return null as well.

    The code is as follows: (see the next page)

```
try{
    // test if lookupLocal() checked the first HashMap
    // lookupLocal() should find ("nameSame"="int") in this case
    // if failed, will output an error message.
    if(!symTable.lookupLocal("nameSame").getType().equals("int")){
        System.out.println(
            "lookupLocal() failed to find the first HashMap in List");
    }

    // test if lookupLocal() can find the correct key
    // lookupLocal() should find ("name21"="char") in this case
    // if failed, will output an error message.
    if(!symTable.lookupLocal("name21").getType().equals("char")){
        System.out.println(
            "lookupLocal() failed to find the first HashMap in List");
    }

    // test if lookupLocal() would return null when key is invalid
    // lookupLocal() cannot find it when key is "nameNotExist"
    // if found, will output an error message.
    if(symTable.lookupLocal("nameNotExist") != null){
        System.out.println(
            "lookupLocal() should return null in this case");
    }

    // test if lookupLocal() would return null when the key is not in
    // the first HashMap
    // if found, will output an error message.
    if(symTable.lookupLocal("name11") != null){
        System.out.println(
            "lookupLocal() should return null in this case");
    }
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

11. We then test the case when attempting to add a duplicate name to the first HashMap. I called addDecl() with its name set as “name21” which already existed. In this case, there will be a DuplicateSymException that we need to catch. I modified the code block so that it catches the exception but does not output anything. The code is as follows:

```
// add a duplicate key to test addDecl()
// this should be an DuplicateSymException, will be caught, no output
try{
    symTable.addDecl("name21", symList.get(7));
    System.out.println("this should not be print");
}
catch(DuplicateSymException ex){
    // Exception caught, no output here
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

12. Next, we test the case when calling addDecl() with either name or Sym is null. Again, there will be an IllegalArgumentException, which will be caught, but no output produced. The code is as follows:

```
// test addDecl() with name is null
// this should be an IllegalArgumentException, will be caught, no output
try{
    symTable.addDecl(null, symList.get(0));
}
catch(IllegalArgumentException ex){
    // Exception caught, no output here
}
catch(DuplicateSymException ex){
    System.out.println("DuplicateSymException occurs");
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

```
// similarly, test addDecl() with Sym is null
// this should be an IllegalArgumentException, will be caught, no output
try{
    symTable.addDecl("namex", null);
}
catch(IllegalArgumentException ex){
    // Exception caught, no output here
}
catch(DuplicateSymException ex){
    System.out.println("DuplicateSymException occurs");
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

13. We could also test the functionality of lookupGlobal().
    - If we call lookupGlobal("nameSame"), it should return Sym of type "int" since it should return the first matched name.
    - If we call lookupGlobal("name22"), which is in the first HashMap, it should return Sym of type "int".
    - If we call lookupGlobal("name12"), which is not in the first HashMap, it should return Sym of type "char".
    - If we call lookupGlobal("nameNotExist"), it should return null.

    The code is as follows:

```
try{
    // test if lookupGlobal() could find the first key "nameSame"
    // in this case, the corresponding value should be "int"
    if(!symTable.lookupGlobal("nameSame").getType().equals("int")){
        System.out.println(
            "lookupGlobal() failed to find the key in first HashMap");
    }

    // test if lookupGlobal() could find "name22" in the first HashMap
    // in this case, the corresponding value should be "int"
    if(!symTable.lookupGlobal("name22").getType().equals("int")){
        System.out.println(
            "lookupGlobal() failed to find the key name22 in symTable");
    }

    // test if lookupGlobal() could find "name12" in the second HashMap
    // in this case, the corresponding value should be "char"
    if(!symTable.lookupGlobal("name12").getType().equals("char")){
        System.out.println(
            "LookupGlobal() failed to find the key name12 in symTable");
    }

    // test if lookupGlobal() could return null when key is not in any
    // HashMap in the symTable
    if(symTable.lookupGlobal("nameNotExist") != null){
        System.out.println(
            "LookupGlobal() failed to return null when key is invalid");
    }
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

14. We then could test removeScope() to see if it successfully removes the first HashMap in symTable. If it does, then we will no longer be able to find the names in the first HashMap. If it does not, then an error message will be produced. The code is as follows:

```
// test if removeScope() could successfully remove the first HashMap
// If it does, then we cannot find the names in the first HashMap.
try{
    symTable.removeScope();
    if(symTable.lookupGlobal("name21") != null
            || symTable.lookupGlobal("name22") != null
            || symTable.lookupGlobal("name23") != null
            || symTable.lookupGlobal("nameSame")
                    .getType().equals("int")){
        System.out.println("removeScope() failed");
    }
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

15. What we need to test next is if addDecl(), lookupLocal(), lookupGlobal() and removeScope() could handle the EmptySymTableException. Before that, we need to call removeScope() to make symTable empty:

```
// test of EmptySymTableException of addDecl(), lookupLocal(),
//                                  lookupGlobal, removeScope()
// firstly, need to call removeScope() to make the symTable empty
try{
    symTable.removeScope();
}
catch(EmptySymTableException ex){
    System.out.println("EmptySymTableException occurs");
}
```

Then, we could test the four functions one by one. Note that, in the catch block which handles the EmptySymTableException, there will be no output.

```
// test of EmptySymTableException of addDecl()
try{
    symTable.addDecl("namex", symList.get(0));
}
catch(EmptySymTableException ex){
    // Exception will be caught, no output
}
catch(DuplicateSymException ex){
    System.out.println("DuplicateSymException occurs");
}
```

```
// test of EmptySymTableException of lookupLocal()
try{
    symTable.lookupLocal("name11");
}
catch(EmptySymTableException ex){
    // Exception will be caught, no output
}
```

```
// test of EmptySymTableException of lookupGlobal()
try{
    symTable.lookupGlobal("name11");
}
catch(EmptySymTableException ex){
    // Exception will be caught, no output
}
```

```
// test of EmptySymTableException of removeScope()
try{
    symTable.removeScope();
}
catch(EmptySymTableException ex){
    // Exception will be caught, no output
}
```

Up till now, all the functionalities of all methods of Sym class and SymTable class have been tested. Recall that only the print() will produce output. Besides that, P1.java produces output only if a test fails. Hence, the expected output will be:

```
Sym Table
{}


Sym Table
{name22=int, name21=char, nameSame=int}
{name12=char, name11=int, nameSame=char}
```