Homework 3a: Review Problems for Exam 2

You don't have to do these problems, and if you do, you don't have to turn them in. However, you can get extra credit points for doing them if you turn them in on WebCT them by Monday, October 22 at 10am.

The two problems in this review are meant to explore the two problems dropped from the first exam in a way that also brings out the main issues in chapter 3. They thus cover the objectives [Concepts] and [UseModels].

Note that these problems are rather more involved than what you would see on a test. Feel free to do just part of the grammar in each problem.

Don't use side effects (assignment and cells) in your solutions.

For all programing tasks, you must run your code using the Mozart/Oz system. For these you must also provide evidence that your program is correct (for example, test cases). Turn in (on WebCT) your code and the output of your testing for all questions that require code.

Be sure to clearly label what problem each area of code solves with a comment.

Don't hesitate to contact the staff if you are stuck at some point.

For review of the material, read sections 2.4.3, 2.3.1, 2.6, and chapter 3 of the textbook [RH04]. Also read our handout "Following the Grammar."

Grammar for the Problems

Consider the grammar in Figure 1 on the following page, which describe data structures (records called abstract syntax trees). These are to be thought of as representing in Oz data the statements and expressions of Oz itself. Thus the grammar in the figure represents the abstract syntax of a sugared version of Oz. For example the following Oz statement:

```
A = B
```

is represented by the record structure:

```
assignStmt("A" varIdExp("B"))
```

Note that the grammar uses Oz strings to represent variable identifiers. A more complex example that illustrates more features of the representation is given in Figure 2 on page 3.

```
⟨Statement⟩ ::= skipStmt
     | seqStmt(⟨List ⟨Statement⟩⟩)
     | localStmt(⟨String⟩ ⟨Statement⟩)
     | assignStmt(⟨String⟩ ⟨Expression⟩)
     | ifStmt(⟨Expression⟩ ⟨Statement⟩ ⟨Statement⟩)
     | caseStmt(⟨Expression⟩ ⟨Pattern⟩ ⟨Statement⟩ ⟨Statement⟩)
     | applyStmt(⟨Expression⟩ ⟨List ⟨Expression⟩⟩)
     | namedFunStmt(⟨String⟩ ⟨List ⟨Pattern⟩⟩ ⟨Expression⟩)
     | inStmt(⟨Pattern⟩ ⟨Expression⟩ ⟨Statement⟩)

⟨Expression⟩ ::= varIdExp(⟨String⟩)
     | atomExp(⟨Atom⟩)
     | numExp(⟨Number⟩)
     | recordExp(⟨Expression⟩ ⟨List ⟨Field⟩⟩)
     | procExp(⟨List ⟨Pattern⟩⟩ ⟨Statement⟩)
     | ifExp(⟨Expression⟩ ⟨Expression⟩ ⟨Expression⟩)
     | caseExp(⟨Expression⟩ ⟨Pattern⟩ ⟨Expression⟩ ⟨Expression⟩)
     | applyExp(⟨Expression⟩ ⟨List ⟨Expression⟩⟩)

⟨Pattern⟩ ::= varIdPat(⟨String⟩)
     | atomPat(⟨Atom⟩)
     | recordPat(⟨Atom⟩ ⟨List ⟨Field⟩⟩)

⟨Field⟩ ::= colonFld(⟨Atom⟩ ⟨Expression⟩)
     | posFld(⟨Exp⟩)
```

Figure 1: Grammar for a simplified subset of the practical version of the declarative language of chapter 3, based on the textbook's section 2.6 [RH04]. The type ⟨String⟩ is the type of character strings in Oz, ⟨Atom⟩ is the type of atoms in Oz, etc.

The following Oz expression:

```
fun {AddToEach A#B Ls}
   case Ls of
     (X#Y)|T then ({Plus A X}#{Plus B Y})|{AddToEach A#B T}
   else nil
   end
end
```

is represented by the following record structure:

```
namedFunStmt("AddToEach"
             [recordPat('#'
                        [posFld(varIdExp("A")) posFld(varIdExp("B"))])
              varIdPat("Ls")]
   caseExp(varIdExp("Ls")
           recordPat('|'
                     [posFld(recordExp(atomExp('#')
                                       [posFld(varIdExp("X"))
                                        posFld(varIdExp("Y"))]))
                      posFld(varIdExp("T"))])
           recordExp(atomExp('|')
                     [posFld(recordExp(atomExp('#')
                                        [posFld(applyExp(varIdExp("Plus")
                                                         [varIdExp("A")
                                                          varIdExp("X")]))
                                         posFld(applyExp(varIdExp("Plus")
                                                         [varIdExp("B")
                                                          varIdExp("Y")]))]))
                      posFld(applyExp(varIdExp("AddToEach")
                                      [recordExp(atomExp('#')
                                                 [posFld(varIdExp("A"))
                                                  posFld(varIdExp("B"))])
                                       varIdExp("T")]))])
           atomExp(nil)))
```

Figure 2: Example showing how Oz is parsed into the record structures (abstract syntax trees) from Figure 1 on the previous page.

Problems

1. (90 points; extra credit) [Concepts] [UseModels]

   Write a function

   ```
   FreeVarIds: <fun {$ <Statement>}: <Set <String>>
   ```

   that takes a ⟨Statement⟩ in the grammar of Figure 1 on page 2 and returns a set of all the variable identifiers that occur free in its argument. You are responsible for writing tests and deciding what the correct outputs for the tests are that match the definition of "free variable identifier." Be sure to include tests that exercise all of the grammar.

   You should use your code for `SetOps` from homework 3 to deal with sets (see our solution on WebCT if you don't have one that works).

   Hint: the result of calling `FreeVarIds` on the record structure in the bottom of Figure 2 on the previous page is a set of strings equivalent to `{AsSet ["AddToEach""Plus"]}`.

2. (90 points; extra credit) [Concepts] [UseModels]

   Write a function

   ```
   BoundVarIds: <fun {$ <Statement>}: <Set <String>>
   ```

   that takes a ⟨Statement⟩ in the grammar of Figure 1 on page 2 and returns a set of all the variable identifiers that occur bound in its argument. You are responsible for writing tests and deciding what the correct outputs for the tests are that match the definition of "bound variable identifier." Be sure to include tests that exercise all of the grammar.

   Hint: the result of calling `BoundVarIds` on the record structure in the bottom of Figure 2 on the previous page is a set of strings equivalent to `{AsSet ["A""B""Ls""X""Y""T"]}`. The variable identifiers `X` `Y` and `T` are declared in the case pattern, and used in that case clause.

   Hint: use `FreeVarIds` and its helping functions (of which you have several, right?) as helping functions.

3. (90 points; extra credit) [Concepts] [UseModels]

   Write a function

   ```
   Desugar: <fun {$ <Statement>}: <Statement>
   ```

   that takes a ⟨Statement⟩ in the grammar of Figure 1 on page 2 and returns a ⟨Statement⟩ in the subset of that grammar that represents Oz statements in the kernel language of the textbook [RH04, Section 2.3]. You are responsible for writing tests and deciding what the correct outputs for the tests are that properly desugar the input. Be sure to include tests that exercise all of the grammar.

References

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.