CSCI 5161: Introduction to Compilers Spring 2023, University of Minnesota Homework 1

Posted: Jan 18, 2023
Due: Feb 3, 2023

Reminder: Assignments are due by 11:59 p.m. on the date indicated.

This assignment is intended to get you started with SML. There are three required parts and one optional part. The required parts get you to do the following in turn:

- Write a little bit of code in the core language.
- Read the first two lectures in Tofte's report and do some simple exercises from there.
- Write a bit of code using functors and signatures that gets you to exercise what you have learned in the second problem.

The directory for hw1 (containing code for the first two problems mainly) is available as a tarball here. You should be thinking of copying this file over, untarring it into a hw1 directory, making your additions to it and then submitting a tarred version of the directory.

The tarred version of the hw1 directory that you submit should contain three code files called slp.sml, SymTblFct.sml and btree.sml as described below and a README file that provides the written answers and/or comments relating to the code for each of the first three problems below.

Problem 1

[The solution to this problem should be contained in a file called slp.sml in the tarball of your hw1 directory.]

This problem requires you to do the exercise at the end of Chapter 1 in the book. In particular, you should implement the two functions *maxargs* and *interp* that the book describes. Write these definitions in functional style. In particular, *the only place where you are allowed to use side effects is in displaying the result of printing*. The starting code for this exercise is provided in the file slp.sml. To get you started, I have added the definition of the table type and the lookup and update functions to the datatype declarations in the book.

There are a few things that are useful to mention before you get going:

- I found some ambiguity in the description of *print* as it appears in the book. The specific point of ambiguity was the following: Suppose the print statement is

  print (e1, e2, e3)

  and that there is also a *print* embedded within *e2*. Should this be performed before or after printing the values of *e1*, *e2* and *e3* or interspersed in some way with this top-level printing? The most sensible interpretation that is consistent with the formatting associated with *print* seems to be to evaluate *e1*, *e2* and *e3* first, carrying out whatever printing is entailed in this process and only in the end to print out the values found for the expressions.

- One would, of course, want to know how one does I/O in ML. The places to find such information are the Standard ML Basis Library and Standard ML of New Jersey Library; one is common to all implementations of SML and the other presents special features of the implementation we are using. Wandering around in the Basis Library a bit, one finds a description of the TextIO structure from where one can quickly locate the print function. You can use this as either *TextIO.print* or simply as *print*.

- Notice that *print* prints strings but we want to print integers. How to accomplish that? If you look some more in the Basis Library, you find the structure Int that has a function *toString* associated with it.

I have provided some example straightline programs and the expected results here; this should help address questions that might arise as you work on the problem. These examples are also available as a tarball here.

Problem 2

[The code for this problem should be contained in the file SymTblFct.sml and any written answers should be provided in the README file, properly marked as answers to the relevant parts of the problem.]

1. Read the first two lectures in Tofte's report. Try the exercises with the first lecture as you go along. After you have tried Exercises 9-11, you may want to compare them with my solutions that are presented in the combination of the SML code that appears here and here. (Nothing needs to be turned in for this part.)

2. From Lecture 2, you should do the exercises 12, 13 and 14. In more detail

   a. For Exercise 12, define the parts described and run the resulting code (that contains the filled out signatures and the updated functor definition) through the SML type checker till it is free of errors. To avoid retyping, here is the code that is already in the report.

   b. For Exercise 13, define the desired nullary functors, add code to create the corresponding structures from them and run this through the SML type checker (i.e. load the file containing this code and modify it till the type checker passes it).

   c. Exercise 14 requires no code but, rather, a textual response.

3. There is a bit of a problem with the way the SymTblFct functor is presented in Tofte's code that is copied over into the skeleton file for this problem. This code includes the signature of the result it is to produce directly in the definition of the functor. The problem with this is that this makes it difficult to define some other functor, e.g., a functor that generates a parser structure, that is parameterized by a symbol table that would be produced by using the SymTblFct functor. To get over this difficulty, define a signature called *SymTblSig* separately and use this to qualify (i.e., to type-check) the structure that is produced by SymTblFct.

4. Explain any special device of the modules language that you need to use to realize the standalone signature for symbol tables that you have defined in the previous part. (Hint: Section 2.5 of Tofte's report talks about

this issue.)

You can turn in the code for parts 2(a), 2(b) and 3 as one one program. However, you must provide separate *written* answers to each of the other questions asked and you must also specifically indicate the part of the code that pertains to each part of the overall problem.

Problem 3

[The solution to this problem should be contained in a file named btree.sml in the tarball of the hw1 directory.]

We would have seen polymorphism in the core language that allows us to define a binary search tree datatype whose elements can be of any type. To do this correctly, we would have also noted that functions such as *insert* had to be parameterized by the "less than" and "equal to" functions. In this problem, we will see another way of realizing such polymorphism---through functors.

Here is what you need to do in this problem; carry out each step and show your answers for each of them separately.

- Define a signature called *BTREE* that matches with binary tree structures that provide a representation for binary trees together with the operations *initTree* that creates a tree with nothing in it, *insert* that inserts a data item in an existing tree, *find* that determines if a data item appears in the tree and *print* that prints out the contents of the tree in sorted order. The tree representation should be abstract and the *BTREE* signature should be able to type binary tree structures for any kind of data item.
- Define a signature called *ITEM* that encapsulates whatever is needed to be known about any particular data kind to create a structure satisfying the *BTREE* signature.
- Define a functor *BTree* that takes a structure satisfying the *ITEM* signature and produces a structure satisfying the *BTREE* signature.
- Define *ITEM* structures for integers and strings and use them with the *BTree* functor to create structures for integer and string binary trees.
- Create specific integer and string binary trees using the structures and functors defined in the previous steps, check if particular items are in these trees and print them out using the print function you have defined with the *BTree* functor.

Note that the data structure that we are really wanting to implement in this problem is a binary *search* tree. Unfortunately, we cannot build the additional ordering requirement that goes with such trees into the type declaration in ML. Instead, we make these invariants of the functions we define, e.g. *insert* makes sure that the ordering requirement is respected after it is done if it was respected before it started and, similarly, *find* uses this ordering information in searching the tree.

Last updated on Jan 8, 2023 by ngopalan atsign umn dot edu
