

CS 558 Homework 1 – due 5pm, Monday, January 15, 2008

Homework must be submitted by mail to `apt@cs.pdx.edu`, with the subject line “CS558 HW1”. All submitted files (five for this assignment) must be sent as plain-text *attachments* to the mail message (the contents of the message itself will be ignored). It is *your* responsibility to submit the homework in the proper format.

All programs mentioned can be downloaded from the course web page.

1. Getting Started (try this *now!*)

Consider the programs `hw1_1.java` and `hw1_1.sml`. Each takes a string parameter w as input, and writes the string `Hello w !` to standard output. The Java version defines a stand-alone application (*not* applet) which can be compiled by typing `javac hw1_1.java`; this produces a class file `Hello.class`, which can be executed by typing something like `java Hello Andrew`. Note that the parameter is given on the execution command line. The ML version defines a function `hi` which can be loaded into the ML interactive loop by typing `use "hw1_1.sml"`; and executed by typing something like `hi "Andrew"`; . Note that the parameter is given as a function argument, in double quotes.

Your task is to alter *each* of these programs so that it *reverses* the characters in w before doing its output. For example, if the input string is “Andrew”, the output of the altered program should be “Hello werdnA!”. Parameter w should be passed just as before. Your extended Java program should be called `sol1_1.java` and should continue to define a class `Hello` with a `main` function. Your extended ML program should be called `sol1_1.sml` and should continue to define a function `hi`. Hint: Before you try to take the string apart by hand, look at the functions available in the standard libraries such as `String`, `StringBuffer` (in Java), and `List` (in ML). If using Moscow ML, remember to use the option `-P full` to make the full libraries available.

Submit (just) the files `sol1_1.java` and `sol1_1.sml`.

2. Expression Interpreter.

Consider a very simple language of expressions, which we’ll call “E1.” Its abstract syntax is given by the following tree grammar:

```
Prog : Program → Exp
Add  : Exp → Exp Exp
Sub  : Exp → Exp Exp
Mul  : Exp → Exp Exp
Div  : Exp → Exp Exp
Int  : Exp → (int)
```

To actually read and write programs in the language, we’ll use a LISP-style concrete syntax that maps directly to the abstract syntax, specified by the following grammar:

```
exp := int
     | '( '+' exp exp ')'
     | '( '-' exp exp ')'
     | '( '*' exp exp ')'
     | '( '/' exp exp ')'

int := digit
     | digit int
```

This problem asks you to modify interpreters for E1, written in Java and ML, living in files called `hw1_2.java` and `hw1_2.sml` respectively. These interpreters read a file containing an E1 program in the LISP-style concrete syntax described above and evaluate the expression, producing an integer result. The Java version takes the filename to be read as an OS command-line argument, e.g., `java Interp foo.e1`; the ML version takes it as a string argument to a top-level function in the interactive loop, e.g., `Interp.interp "foo.e1"`.

For example, if `foo.e1` contains the following text:

```
(/           { Integer division rounds results down }
 (+ 7
  (- 0 2)) { Here's how to make a negative number }
 3)
```

then both interpreters should return the answer 1. Note that programs can be broken arbitrarily across multiple lines. Any text within curly braces will be treated as a comment.

(a) Add a new feature to each interpreter: support for a remainder (%) operator. The relationship of this operator to the existing operators is defined by the following equation: $(a/b)*b+(a\%b) = a$, for all a and b , regardless of sign.

Note: The ML version does not evaluate the program AST directly; instead it compiles it into a sequence of stack machine instructions, and then executes the resulting stack machine program. In adding support for remainder, you must *not* change the instruction set or implementation of the stack machine.

(Hint: Look at the code for an existing binary operator, e.g., `+`. In the Java version, you need to add a new subclass of `Exp`. In the ML version, you will need to add new clauses to several case statements. In each, you'll need to make two small additions to the parser.)

Put your Java solution into a Java file `so11_2.java`. Put your ML solution into an ML file `so11_2.sml`. Submit these two files.

(b) The ML version and the Java version behave inconsistently on some programs involving large integers. Find out what the differences are, and why they occur. Write a simple program that behaves differently under the two interpreters. Submit this program, and a comment explaining the differences and the reasons for them, in file `so11_2.e1`. (Note: There are several differences. Your sample program may illustrate only one of them, but your comment should describe all that you find.)

WARNING: Do NOT attempt to do this exercise using SML of New Jersey under Solaris; that implementation has bugs that will confuse you. Use `mosml` and/or run on an X86-based machine instead.