What to turn in

Turn in the written parts of the assignment during class on the due date. For the programming part, you should mail your version of the file `interpretation.sml` to `siggi@cs.cornell.edu` by 5PM on that day.

Note: For induction proofs, clearly specify what property you prove and what kind of induction you use.

Problem 1: Command Equivalence

a) Show that the **IMP** while command **while** b **do** c is equivalent to the following command:

if b **then**
  c ;
  while b **do** c
else skip

b) Show that the **IMP** while command **while** b **do** c is equivalent to the following command:

while b **do**
  c ;
  if b **then** c **else skip**

Problem 2: Nondeterministic Choice

Consider a language of nondeterministic expressions with the following gammar:

$$a ::= n \;\mid\; a_0 + a_1 \;\mid\; a_0 \;[]\; a_1$$

where $n \in \mathbb{Z}$ is an integer constant, $a_0 + a_1$ is the sum of two expressions, and $a_0 \;[]\; a_1$ is an expression which nondeterministically evaluates either a_0 or a_1, and yields the value of that expression. We denote by **Nexp** the set of all nondeterministic expressions. The large step operational semantics for this language can be expressed using an evaluation relation $\rightarrow \in \mathbf{Nexp} \times \mathbb{Z}$, defined as follows:

$$\frac{}{n \rightarrow n} \qquad \frac{a_0 \rightarrow n_0 \quad a_1 \rightarrow n_1}{a_0 + a_1 \rightarrow n}\; n_0 + n_1 = n \qquad \frac{a_0 \rightarrow n_0}{a_0 \;[]\; a_1 \rightarrow n_0} \qquad \frac{a_1 \rightarrow n_1}{a_0 \;[]\; a_1 \rightarrow n_1}$$

We want now to give a small step semantics for this language, using a relation $\rightarrow_1 \in \mathbf{Nexp} \times \mathbf{Nexp}$. A single evaluation step $a \rightarrow_1 a'$, reduces expression a to a simpler expression a'; eventually, after multiple steps, an expression a evalutes to a number n. We denote by $\rightarrow_1^*$ the reflexive, transitive closure of this small step evaluation relation. We want to define the small step sematics such that it is equivalent to the above large step semantics:

$$\forall a, n \;.\; a \rightarrow n \Leftrightarrow a \rightarrow_1^* n$$

2a) Consider that we define the small step semantics $\rightarrow_1$ using the following rules:

(1) $$\frac{}{n_0 + n_1 \rightarrow_1 n} \text{ where } n = n_0 + n_1$$

(2) $$\frac{a_0 \rightarrow_1 a_0'}{a_0 + a_1 \rightarrow_1 a_0' + a_1}$$

(3) $$\frac{a_1 \rightarrow_1 a_1'}{a_0 + a_1 \rightarrow_1 a_0 + a_1'}$$

(4) $$\frac{a_0 \rightarrow_1 a_0'}{a_0 \;[]\; a_1 \rightarrow_1 a_0'}$$

(5) $$\frac{a_1 \rightarrow_1 a_1'}{a_0 \;[]\; a_1 \rightarrow_1 a_1'}$$

Is this small step sematics equivalent to the above defined large step semantics? If yes, give a formal proof. If no, give a counter example.

2b) Consider an alternative small step semantics $\to_1$ which we define using the first three rule above, but we replace rules (4) and (5) with the following:

$$(6) \quad \frac{a_0 \to_1 a_0'}{a_0 \, [] \, a_1 \to_1 a_0' \, [] \, a_1}$$

$$(7) \quad \frac{a_1 \to_1 a_1'}{a_0 \, [] \, a_1 \to_1 a_0 \, [] \, a_1'}$$

$$(8) \quad \frac{}{n_0 \, [] \, n_1 \to_1 n_0}$$

$$(8) \quad \frac{}{n_0 \, [] \, n_1 \to_1 n_1}$$

Is this small step sematics equivalent to the large step semantics? If yes, give a formal proof. If no, give a counter example.

Problem 3: Expressions with Side-Effects

In this problem we extend the **IMP** language with more complex expressions and commands, namely expressions that involve computation and thus have side-effects. For this, we first extend the syntax of arithmetic expressions presented with one more production, where commands are treated as arithmetic expressions:

$$a \quad ::= \quad \ldots \quad | \quad c \tag{1}$$

Informally, the meaning of a command as an expression is defined as follows.

- if $c \equiv$ **skip** then the evaluation of a results in an error;
- if c is an assignment $X := a'$ then the value of a is that of a';
- if c is an if statement **if** b **then** c' **else** c'', then the value of a is the value of c' if b evaluates to true, and the value of c'' if b evaluates to false.
- if c is a while loop **while** b **do** c', then the value of a is the value of c' on the last iteration of the loop. If no iteration is executed, the evaluation of the c as an expression results in an error.
- if c is a sequence of commands, then the value of a is the value of the rightmost command in the sequence which is not a **skip** command and is not a **while** whose condition evaluates to **false**. If that command yields an error, then c also produces an error.

The following program uses commands as expressions to iteratively compute the factorial of `X` if `X` is positive. Otherwise, the program enters an error state.

```
X := 5;
ANSWER := {
   if (0 <= X) then {
      Y := 1;
      while (2 <= X)
        { Y := Y * ({X := X-1} + 1) }
   } else {
      skip
   }
}
```

$$
\begin{array}{rcl}
aexp & ::= & n \mid \texttt{X} \mid aexp + aexp \mid aexp * aexp \mid aexp - aexp \mid (\ aexp\ ) \mid \{\ com\ \} \\[1em]
bexp & ::= & \texttt{true} \mid \texttt{false} \mid aexp = aexp \mid aexp <= aexp \mid \texttt{not}\ bexp \\
 & & \mid bexp\ \texttt{or}\ bexp \mid bexp\ \texttt{and}\ bexp \mid (\ bexp\ ) \\[1em]
com & ::= & \texttt{skip} \mid \texttt{X} := aexp \mid \texttt{if}\ (\ bexp\ )\ \texttt{then}\ \{\ com\ \}\ \texttt{else}\ \{\ com\ \} \\
 & & \mid \texttt{while}\ (\ bexp\ )\ \{\ com\ \} \mid com\ ;\ com \mid aexp
\end{array}
$$

Figure 1: Concrete Syntax for IMP + Side Effects

Note that the `if` and the `while` statements in this program are both expressions. Since `X` has a positive value, the value of the `if` statement is the value of its true branch; the value of the `while` loop is the value being assigned to `Y` in the last iteration of the loop. Note that for `X = 0` or `X = 1`, the while loop does not execute any iteration, so the value of the sequence `Y := 1; while ...` is given by the assignment `Y := 1`.

We next extend the grammar of commands, and allow each expression to be a possible command:

$$c \quad ::= \quad \dots \mid a \tag{2}$$

The execution of such a command just evaluates a and records the side effects; if the command c is further used as an expression, the value of that expression is the value of a. This makes it easier to return values on the branches of if commands without creating temporary variables. For instance, the following program uses this extension to directly compute the maximum of three given values, `X`, `Y`, and `Z`.

```
X := 5; Y := 1; Z := 2;
ANSWER := {
   if (X <= Y) then {
      if (Y <= Z) then { Z } else { Y }
   } else {
      if (X <= Z) then { Z } else { X }
   }
}
```

The concrete syntax of IMP with commands as expressions and expressions as commands is shown in Figure 1. It is essentially the abstract syntax discussed in class, augmented with parentheses and C-like syntax for `if` and `while` commands. Locations are denoted by strings containing only upper or lower case alphabetic characters. Keywords are all in lower case. The arithmetic operators have the usual precedence and associativity, and you can put parentheses in to group them. Note that `;` acts as a command separator not a terminator, so you will get parse errors if you put `;` at the end of a sequence. Finally, the command separator `;` is left-associative, although the grammar doesn't show that.

(3a) **Operational Semantics**

Define the large-step operational semantics for **IMP+SE** (IMP with SideEffects) according to the informal description above. To capture the fact that the evaluation of arithmetic expressions and boolean expression may have side effects, you must use evaluation relations that yield a value and a new state:

$$
\begin{array}{rl}
\rightarrow_a \subseteq (\mathbf{Aexp} \times \Sigma) \times (\mathbb{Z} \times \Sigma) & \quad \langle a, \sigma \rangle \rightarrow_a \langle n, \sigma' \rangle \\
\rightarrow_b \subseteq (\mathbf{Bexp} \times \Sigma) \times (\mathbb{T} \times \Sigma) & \quad \langle b, \sigma \rangle \rightarrow_b \langle t, \sigma' \rangle
\end{array}
$$

The form of the evaluation relation for commands is the same as for **IMP**:

$$\rightarrow_c \subseteq (\mathbf{Com} \times \Sigma) \times \Sigma \qquad \langle c, \sigma \rangle \rightarrow_c \sigma'$$

You must define all of the above relations ($\rightarrow_a$, $\rightarrow_b$, and $\rightarrow_c$) for **IMP+SE**. For clarity, you are required to explicitly use the subscripts for the evaluation relations (because a command can now be executed using $\rightarrow_c$, or evaluated to a number using $\rightarrow_a$). Write your semantics such that the evaluation gets stuck when it reaches an error case.

Most of the rules that define the operational semantics are straightforward extensions of the **IMP** semantics. The novel part is expressing the evaluation of commands as arithmetic expressions, and, in particular, the evaluation of while loops and sequencing of commands. You can assume that the sequencing of commands is left-associative: if $c_0; c_1$ is a command sequence, then c_1 is not a sequence of commands. This will make it easier to write the rules for the evaluation of sequences as expressions.

(3b) **Implementation**

The code supporting this assignment provides a lexer and a parser for standard **IMP**. It also provides an interpreter based on the operational semantics of **IMP**, as discussed in class. The interpreter can be found in the source file `interpretation.sml`. This file implements the evaluation of commands in standard **IMP** as a function `interpret`, which, in turn, invokes two other functions, `interpretAexp` and `interpretBexp`, for the evaluation of expressions.

You are required to implement a new interpretation function, `interpretSE`, for the evaluation of commands in **IMP+SE**, according to the operational semantics defined in (3a). Your implementation will raise an exception when the execution reaches an error case. You will also need to implement two other functions, `interpretAexpSE` and `interpretBexpSE`, for the the evaluation of arithmetic expressions and boolean expressions in **IMP+SE**. Use the structure of the interpretation functions for **IMP** as a model when implementing the interpretation of **IMP+SE**. The only file you need to modify is `interpretation.sml`.

Getting started

For the implementation part, you should download the file `imp.tar.gz` from the CS611 web page for Homework 1 and extract the contents to a working directory.

These files contain the source code to a simple IMP interpreter (the file `interpretation.sml`) as discussed in class. To make it easier for you to debug and test your code, we have written a lexer and parser (the `imp.lex` and `imp.grm` files). We've also included a pretty-printer for the abstract syntax (`pprint.sml`). The file `top.ml` contains two functions, `test` and `testSE`, both of type `string -> int` that take the name of a file containing IMP code, pretty-print it, interpret the program, and print out the result contained in the `ANSWER` location. The `testSE` function will be used to run **IMP** programs extended with the side-effect expressions as described above. Both of these functions can be called at the prompt in the SML system. Feel free to browse through the source code provided, but for the purposes of this assignment, *the only file you should modify is* `interpretation.sml`. The other code makes use of sml module system, but you shouldn't need to know how it works to do this assignment.

Be sure `sml` is in your path. On the machines in the Linux cluster and on the Solaris machines in the CS department, it can be found at `/home/sml/bin/sml`. If you want to install it on your own machine, you can download the appropriate version (including the Windows version) from the SML home page : `http://cm.bell-labs.com/cm/cs/what/smlnj`.

To compile your IMP interpreter, first untar `imp.tar.gz` and then edit the file `sources.cm` to set the appropriate path to the ML-Yacc libraries. Next, start SML/NJ in the directory with the source code and type `CM.make()` at the prompt. The first time you do this, all of the files will be compiled and cached on disk. From then on, you can recompile changes you've made by doing `CM.make();` again – only those files that have changed will be recompiled. `CM.make()` also adds the toplevel function definitions to the environment. At this point you should be able to test the file `test1.imp` by typing `test "test1.imp"`.