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 `fluet@cs.cornell.edu` by 5PM on that day.

1. Term Equivalence

   Consider a version of IMP in which we add the construct **do** c **until** b. We have some choice about how to extend the (large-step) semantics. One option is to notice that this statement is equivalent to the statement c; **while** $\neg b$ **do** c, and write an corresponding inference rule:

   $$\frac{\langle c;\ \textbf{while}\ \neg b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{do}\ c\ \textbf{until}\ b, \sigma\rangle \Downarrow \sigma'}$$

   (a) Another option is to express the large-step semantics of **do** ... **until** directly, without building on **while**. Give two inference rules that do this.

   (b) Prove that in the extended language based on the large-step semantics of part 1(a), the command **do** c **until** b and the command c; **while** $\neg b$ **do** c are equivalent in all states and for all commands c and boolean expressions b. *Hint:* expand their proof trees.

2. Structural induction

   (a) Exercise 3.5, Winskel. You are asked to show that the large-step evaluation *relation* is actually a *function*, and that it is defined on its entire domain (that is, total).

   (b) Exercise 2.4, Winskel.

   (c) Because of the result in part 2(a), the semantics you have defined for part 2(b) don't actually differ from the left-to-right evaluation semantics for disjunction. Informally describe a language construct that you could add to IMP that would make the two semantics differ.

   (d) Use structural induction to show that the two semantics are identical in standard IMP. You may assume that the result of part 2(a) holds for both semantics even though you have shown it holds only for the left-to-right evaluation order.

3. Adding **break**

   IMP is lacking many features of a real programming language, most notably functions and data structures, which we will study later. In this problem you will add a mechanism like C's **break** statement that allows a program to terminate **while** loops early. While this is a seemingly minor change to the language, we shall see that it radically alters the semantics of IMP.

   As an example, the program in Figure 1 (also found in `testB1.imp`) should halt with the value 8 in location `ANSWER`:

   This example brings out several points when adding **break** to IMP. First, if the **break** occurs in a sequence of commands, the remainder of the sequence (in this case `ANSWER := ANSWER + 10`) is not evaluated. Second, there must be a way to keep track of where the program will continue after evaluating a **break** command. Since **while** loops may be nested, these **break** continuations form a stack.

   A **continuation** can be thought of as "the rest of the evaluation of the program", and we shall study them in more depth later. For now, we can think of continuations as "the next command to be executed", which, in general, will be a sequence representing *all* of the commands remaining in the program. As we observed above, introducing **break** requires that we have the ability to both discard the continuation of a command in a sequence and save a continuation on a stack so we know where to pick up execution after evaluating **break**. This suggests that we modify IMP's program configurations

```
while (true) {
  while (true) {
    ANSWER := ANSWER + 1;
    if (ANSWER = 4) then {
      break;
      ANSWER := ANSWER + 10
    } else {
      skip
    }
  };
  ANSWER := ANSWER * 2;
  break
}
```

Figure 1: Sample program using **break**

$$
\begin{array}{rcl}
aexp_c & ::= & n \mid \text{Loc} \mid aexp_c + aexp_c \mid aexp_c * aexp_c \mid aexp_c - aexp_c \mid (\ aexp_c\ ) \\
\\
bexp_c & ::= & \texttt{true} \mid \texttt{false} \mid aexp_c = aexp_c \mid aexp_c <= aexp_c \mid \texttt{not}\ bexp_c \\
& & \mid bexp_c\ \texttt{or}\ bexp_c \mid bexp_c\ \texttt{and}\ bexp_c \mid (\ bexp_c\ ) \\
\\
com_c & ::= & \texttt{skip} \mid \texttt{break} \mid \text{Loc} := aexp_c \mid \texttt{if}\ (\ bexp_c\ )\ \texttt{then}\ \{\ com_c\ \}\ \texttt{else}\ \{\ com_c\ \} \\
& & \mid \texttt{while}\ (\ bexp_c\ )\ \texttt{do}\ \{\ com_c\ \} \mid com_c\ ;\ com_c
\end{array}
$$

Figure 2: IMP concrete syntax

to be the following: $\langle c_{cur}, c_{next}, \rho, \sigma\rangle$, where c_{cur} is the command currently being executed, c_{next} is the continuation of c_{cur}, ρ is a stack of commands, written $c_n :: c_{n-1} :: \ldots :: c_1 :: []$, each c_i representing the continuation of any **break** commands in the body of a **while** loop with nesting depth i, and σ is an IMP state as usual.

There are a few subtle points to address. The first question is: What happens when a **break** command is executed outside of any **while** -body?, that is when $c_{cur} =$ **break** and $\rho = []$. This can be considered an ill-formed program which should cause an error to occur—our interpreter will model this by raising the exception `BadBreak` defined in `interpretation.sml`. The second question is: What is the continuation of a single-command program such as `X := 3`? Since such an IMP program should halt after executing the command, we could add a special `Halt` continuation to the interpreter, but to make things easier, we will adopt the convention that if the **skip** command appears as c_{next} then the program is to terminate after evaluating c_{cur}. (This makes the interpreter slightly simpler to write.)

With these observations in hand, we can now specify the operational semantics of IMP+**break**. The new judgements will be of the form $\langle c_{cur}, c_{next}, \rho, \sigma\rangle \rightarrow \sigma'$. As an example, consider the following inference rule specifying the behavior of the assignment command:

$$\frac{\langle a, \sigma\rangle \rightarrow n \quad \langle c_{next}, \textbf{skip}, \rho, \sigma[n/X]\rangle \rightarrow \sigma'}{\langle X := a, c_{next}, \rho, \sigma\rangle \rightarrow \sigma'}$$

To evaluate the command $X := a$ followed by c_{next} with **break** continuations ρ and state σ, we first evaluate the arithmetic expression a in state σ to obtain the number n. Next we evaluate the continuation c_{next} with **skip** as its continuation (indicating the program should then halt), the same ρ, and a new state obtained from σ by updating X to be n.

Here is the axiom for specifying when a program halts:

$$\frac{}{\langle \textbf{skip}, \textbf{skip}, [], \sigma\rangle \rightarrow \sigma}$$

Here is the rule for evaluating a **skip** command as part of a sequence of other commands:

$$\frac{\langle c_{next}, \textbf{skip}, \rho, \sigma\rangle \rightarrow \sigma'}{\langle \textbf{skip}, c_{next}, \rho, \sigma\rangle \rightarrow \sigma'} \ (c_{next} \neq \textbf{skip})$$

(a) Operational Semantics

Define the full set of operational semantics for IMP+**break** using the rules above as guidelines. You will need to define the proper behavior for the interaction of **break** and **while** statements. In particular, **break** should discard c_{next} and **while** should save its continuation on the top of ρ. Be careful!

(b) Implementation

The lexer and parser have already been modified to support the **break** command. Implement the interpreter for IMP+**break** as the function `interpretBreak : configBreak -⟩ state` found in the source file `interpretation.sml`. The stack of continuations, ρ, is implemented by an `ImpAst.com list`. This should be straightforward, once you've decided what the operational semantics should be. Conversely, if you can implement the interpreter correctly, you should be able to write down its operational semantics. The only file you need to modify is `interpretation.sml`.

Note that on programs not containing the **break** command, `test` and `testB` should agree. Feel free to post interesting test cases to the newsgroup.

Getting started

For problem 3, 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.sml` contains two functions, `test` and `testB`, 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 `testB` function will be used to run IMP programs extended with a `break` construct as described below. There are also a couple example IMP programs (`*.imp`).

Feel free to browse through the source code provided, but for the purposes of this assignment, *the only file you have to modify is* `interpretation.sml`. The other code makes use of ML's module system, but you shouldn't need to know how it works to do this assignment.

To compile your IMP interpreter, first edit the file `sources.cm` to set the appropriate path to the ML-Yacc libraries. (For the appropriate path, see the comments in `sources.cm`.) 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 run the file `test1.imp` by typing `test test1.imp` at the prompt.

The concrete syntax of IMP is shown in Figure 1. It is essentially the abstract syntax discussed in class augmented with parenthesis and C-like syntax for `if` and `while` commands. Locations are denoted by strings containing only upper or lower case alphabetic characters. Negative numbers are prefixed with `~` as in ML. Keywords are all in lower case. The arithmetic operators have the usual precedence and associativity, and you can put parenthesis 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.