CS611 Homework 1 DUE: 09/17/01

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.ml` and one interesting test case to `nystrom@cs.cornell.edu` by 5PM on that day.

1. Term equivalence

   (a) Using the large-step operational semantics of IMP, prove that the statement **if** b **then** c_0 **else** c_1 is equivalent to **if** $\neg\, b$ **then** c_1 **else** c_0.

   (b) Similarly, prove that command sequences in IMP can be either left or right associative by showing that $(c_0;\ c_1);\ c_2$ is equivalent to $c_0;\ (c_1;\ c_2)$

2. Auto-increment expressions

   Suppose we add C-like auto-increment expressions to IMP. Given a variable x, x++ will increment x by 1 and evaluate to the previous value of x; ++x will increment x by 1 and evaluate to the new value of x. For example,

   $$\texttt{ANSWER} := 1;\ \textbf{while}\ \texttt{i++} <= 4\ \textbf{do}\ \texttt{ANSWER} := \texttt{ANSWER} * \texttt{i}$$

   stores 120 in the variable `ANSWER`, while

   $$\texttt{ANSWER} := 1;\ \textbf{while}\ \texttt{++i} <= 4\ \textbf{do}\ \texttt{ANSWER} := \texttt{ANSWER} * \texttt{i}$$

   stores 24 in the variable `ANSWER`.

   (a) Write inference rules for the large-step operational semantics for the two auto-increment expressions.

   (b) Since expressions can now modify the store, the form of the evaluation relations ($\Downarrow$) for arithmetic and boolean expressions must change. Explain how the form of the relations changes and describe how to rewrite the inference rules for both expression types. You do not have to write the new inference rules.

   (c) Another issue that arises when expressions can have side effects is that their order of evaluation now matters. The operational semantics for IMP presented in class do not generally specify order of evaluation. For example, the inference rule in IMP for relational expressions:

   $$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \quad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 <= a_1, \sigma\rangle} \text{ where } t \text{ is true iff } n_0 \leq n_1$$

   does not specify which of a_0 and a_1 is evaluated first. Thus, if x is initialized to 0, the expression x++ $<=$ x may evaluate to either true or false. With these semantics, IMP with auto-increment is non-deterministic. Write the *small-step* operational semantics of IMP with auto-increment, being sure to define the language so that it is deterministic.

   (d) Use induction on derivations with the semantics you wrote for part 2(c) to prove that arithmetic expressions are deterministic.

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 will see that it radically alters the semantics of IMP.

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

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

Figure 1: IMP+**break** concrete syntax

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

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 will 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 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 this: 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.ml`. A **break** command which attempts to terminate more more loops than the current nesting level will similarly fail. The second question is this: 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 $\langle c_{cur},\, c_{next},\, \rho,\, \sigma\rangle \Downarrow \sigma'$. As an example, consider the following inference rule specifying the behavior of the assignment command:

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

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:

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

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

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

(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 : breakConfig -> state` found in the source file `interpretation.ml`. The stack of continuations, ρ, is implemented by an `Ast.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 should modify is `interpretation.ml`. No other file should be modified.

Note that on programs not containing the **break** command, `test` and `testB` should agree. Feel free to post test cases to the newsgroup. When you email in your source code, you should submit one interesting test case of your own devising along with your modified `interpretation.ml`.

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.ml`) as discussed in class. To make it easier for you to debug and test your code, we have written a lexer and parser (the `lexer.mll` and `imp.mly` files). We've also included a pretty-printer for the abstract syntax (`pprint.ml`). The file `main.ml` 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. Both of these functions can be called from the top-loop of the ocaml system. There are also a couple of example IMP programs (`*.imp`).

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

To compile your IMP interpreter, first untar `imp.tar.gz`. Be sure `ocaml` is in your path. On the Solaris machines in the CS department, it can be found at `/usr/u/plg/bin/sparc-solaris`. On the Windows machines in the CS department, we have placed a copy in `\\bigbird\users\nystrom\ocaml`, or `z:\nystrom\ocaml` if `\\bigbird\users` is mounted as usual. You can download your own copy of OCaml from the Caml web page at `http://caml.inria.fr`. You make have to adjust the `Makefile` or, on Windows, `make.bat` to point to your installation of OCaml. Now, run `make`. This will run a batch file on Windows and invoke `make` on Unix. After the build completes, you should have an executable named `imp`. At this point you should be able to run `ocamlrun imp test1.imp` at the prompt. This will invoke the interpreter for unmodified IMP. Running `ocamlrun imp -break test1.imp` will invoke your interpreter for IMP+**break**. This should display the error message “Bad break.”

The concrete syntax of IMP 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. 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 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.