CS411 Probs 2

9 Feb 2001 - due 20 Feb 2001

For this problem set, you will be asked to work with an interpreter for **IMP** that we have provided for you on the web site. The interpreter is implemented in SML. To use the interpreter, download the file `imp.sml`, put it in some working directory, fire up SML/NJ, set the working directory,[1], then evaluate `use "imp.sml"`. This loads the structure `Imp`. (Note that everytime you modify `imp.sml`, you will need to *use* it again if you want to test your modifications.) To start the interpreter, evaluate `Imp.run ()`. Try out some **IMP** expressions. For example, `X <- 10`, or `Y <- 25; skip; if true then X <- 1 else X <- 2`. (Syntax-wise, all keywords are lowercase, assignment is written `<-`, the boolean NOT operator is written `!`, the boolean AND operator is written `&&`, and the boolean OR operator is written `||`.) Note that after every evaluation, the state is printed. Every evaluation occurs in the state as returned by the previous evaluation.

Download and study the interpreter. Structurally, it is straightforward. It is simply a loop, that awaits a **IMP** command from the user, parses it into abstract syntax,[2] and applies the small-step semantic reduction rules repeatedly until the empty command signaling termination is encountered.

The abstract syntax is represented by values of a datatype (three datatypes in fact, `com`, `aexp`, `bexp`, representing the three syntactic classes of **IMP**). You will want to review Harper's notes on datatypes. Because of this representation using datatypes, the functions to apply reduction rules (`applyComRule`, `applyAExpRule`, `applyBExpRule`) use pattern matching to figure out which rule to apply. Again, you will want to review Harper's notes on pattern matching. Note that the order of the patterns is important. The first pattern that matches is selected. Look at the implementation of the rules for arithmetic expressions to see an illustration of this. In general, you will want to compare the small-step reduction rules of **IMP** with the code, to see how we implemented the reductions.

For this problem set, you will be asked to add new commands to the interpreter, by adding code to implement new reduction rules. The interpreter already knows about the new commands. So you don't have to worry about parsing or anything like that. However, the interpreter cannot evaluate (or reduce) such commands. That's where you come in.

[1] If you stored the file in directory `c:\foo\bar`, change the working directory of SML/NJ by evaluating `OS.FileSys.chDir "c:\foo\bar"`. Use forward slashes under Unix, as usual.

[2] The details of the parser are beyond the scope of this course, so don't worry about it. It's a monadic parser, if you're curious.

1. Once a language has while-loops, it is a simple matter to add support for repeat-loops. Consider extending **IMP** with a new command:

   $$c ::= \cdots \mid \textbf{repeat } c \textbf{ until } b$$

   Intuitively, this says to repeatedly execute c until the condition b evaluates to true. We can provide a small-step semantics for this rule by doing the same kind of trick we did for the while-loop: rewrite it into an equivalent command in the language. You can check that the following reduction rule does what we want:

   $$\frac{}{\langle \textbf{repeat } c \textbf{ until } b, \sigma\rangle \rightarrow_1 \langle c; \textbf{while } \neg b \textbf{ do } c, \sigma\rangle}$$

   The **IMP** interpreter already supports the syntax for repeat-loops. Implement the above reduction rule in the interpreter.

2. Here is an extension of **IMP** that models a form of non-local control transfer, like exceptions (in Java or ML) or setjmp/longjmp (for the C hackers).

   We add the following two rules to the **IMP** syntax for commands:

   $$c ::= \cdots \mid \textbf{try } c_0 \textbf{ catch } c_1 \mid \textbf{throw}$$

   Usually, a try-block command of the form **try** c_0 **catch** c_1 is equivalent to c_0; the associated *catch phrase* c_1 is not executed. However, if a **throw** command is executed anywhere inside c_0, control proceeds to the catch phrase c_1 of the innermost enclosing try-block. For example, the command:

   $$X \leftarrow 1; \textbf{try } Y \leftarrow 2; X \leftarrow 2 \textbf{ catch skip}$$

   results in a state where both X and Y are 2. On the other hand:

   $$X \leftarrow 1; \textbf{try } Y \leftarrow 2; \textbf{throw}; X \leftarrow 2 \textbf{ catch skip}$$

   results in a state where X is 1 and Y is 2, and:

   $$X \leftarrow 1; \textbf{try } Y \leftarrow 2; \textbf{throw}; X \leftarrow 2 \textbf{ catch } Y \leftarrow 1$$

   results in a state where X is 1 and Y is 1.

   Write small-step semantics reduction rules for commands **try** c_0 **catch** c_1 and **throw**.

   The **IMP** interpreter already supports the syntax for such commands. Implement your reductions rules in the interpreter.