Assignment 2:
Implementing MinML

15-312: Foundations of Programming Languages
Daniel Spoonhower (`spoons+@cs.cmu.edu`)

Revised: September 9, 2004
Due: Thursday, September 23, 2004 (11:59 pm)

100 points total

1 MinML

For this assignment you will implement a typechecker and evaluator for MinML. This document, along with an archive including your starting code and some examples, can be found both on the web and via AFS.

```
http://www.cs.cmu.edu/~fp/courses/312/assignments/asst2/
/afs/andrew/scs/cs/15-312/assignments/asst2/
```

In the archive, you will find several files with support code; you will only need to fill in the missing code in `translate.sml`, `typing.sml`, and `eval.sml`.

You will rarely, if ever, need to write long or complicated functions to complete this assignment. Therefore, you should strive for elegance. Your solution will be graded primarily on correctness, but if your code does not correctly handle one or more cases, we will inspect your code and attempt to give you some credit for the understanding it reflects. You will also have the opportunity to reuse your solution to this assignment in future assignments. In each of the latter situations, it is to your benefit to write clean, legible code.

Before you begin, you may wish to read through the provided code (especially the signatures) to gain an understanding of the setup. All of the necessary SML files are listed in the `sources.cm` file, and you can build the project in SML/NJ by typing `CM.make()`.

1.1 Parser and Concrete Syntax

The file `parse.sml` contains a parser for MinML. The `parse` function turns a `Lex.token Stream.stream` into a `MinML.exp Stream.stream` by consuming programs (which are expressions followed by a semicolon). For simplicity, we don't do any error recovery; when the parser encounters an error it just raises the exception `Parse.Error` with a (somewhat) informative message.

While we have written the parser for you and the code you write will deal only with abstract syntax, you still need to know the concrete syntax to write test programs. A grammar is given in Figure 1. The grammar refers to tokens such as `INT` and `BOOL`. The tokens are defined in `lexer.sml`; we have also provided a lexer that takes a raw character stream and returns a stream of tokens.

This syntax should be mostly self-explanatory. Application of a function e_1 to an argument e_2 is written by juxtaposition ($e_1\,e_2$). Primitive operations are infix, with the usual precedence levels (negation has the highest precedence, followed by juxtaposition (for function application), then multiplication, then addition and subtraction, and finally equality). All primitive operations are left-associative.[1] The type constructor '`->`' is infix and right-associative, just as in SML.

We have deviated from the MinML syntax given in Harper's notes at one significant point. Harper uses an expression of the form `fun f(x:t1):t2 is e end` to define *all* functions, whether they are recursive or not. Instead, we have separated the notion of a function from that of a recursive structure: we use `fn x : t => e` to describe functions (written $\lambda x : \tau . e$ in the simply-typed λ-calculus) and `rec x : t => e` to define a recursive term (sometimes seen as $\mu x : \tau . e$). For now, the only (useful) recursive constructs in MinML are recursive functions, but later assignments will offer additional examples.

In order to make MinML programs easier to write and to read, we have introduced a bit of *syntactic sugar* to emulate Harper's `fun`. By syntactic sugar, we mean a feature that is added only to the concrete syntax of a language and has no bearing on the abstract syntax. In order to give meaning to this new syntax, we define a translation into another, existing form of concrete syntax.

$$\texttt{fun f(x:t1):t2 => e} \longleftrightarrow \texttt{rec f:t1 -> t2 => fn x:t1 => e}$$

Syntactic sugar allows us to make the language more user-friendly without complicating the implementation or our attempts to reason about the language. Since your portion of the implementation will deal only with abstract syntax, you won't need to worry about `fun` or any other syntactic sugar (again, except when you write test code).[2]

Figure 2 gives some examples of concrete syntax along with their translation into MinML abstract syntax (SML expressions of type `MinML.exp`). Note that while the concrete syntax found in each of the last two rows is distinct, the abstract syntax is identical. Also, note that the abstract syntax groups binders together with their scope in the style of higher-order abstract syntax. Variables are represented via their name as a string.

To play around with the parser and become familiar with MinML, at the SML/NJ prompt type `Top.loop_print_noDB ();` or `Top.file_print_noDB "test_file.mml";`. These will print the program (with some redundant parentheses) in the named-variable form.

Task: Translation to deBruijn form (20 points)

In the file `translate.sml`, complete the implementation of function `Translate.translate`. When completed, it should translate a stream of closed MinML expressions in the named variable representation (type `MinML.exp`) to a stream of closed expressions in the deBruijn representation (type `DBMinML.exp`). (*Hint:* Use the function `Stream.map`.)

To get started, read Section 5.4 of Harper's notes, then think about how to translate the abstract syntax. Most cases are very straightforward; variables, `let`, `rec`, and `fn` will require more thought. You are free to consider it an error if your translator is fed an open expression, i.e. one

[1] The grammar is actually *right*-associative (since that's easier to implement in a recursive-descent style) so we had to do some work in the parser to produce the correct abstract syntax. If you're interested, look at the `parse_exp`, `parse_exp'`, `parse_term`, `parse_factor`, `parse_factora` and `build_primop` functions in `parse.sml`.

[2] It should be further noted that the `fun` in the MinML of Harper's notes and the one described here both differ from the SML `fun` construct. In both instances of MinML, `fun` denotes only a value and not a declaration, as it does in SML. Our `fn` is, however, very similar to the SML expression with the same name.

```
BaseType ::= INT | BOOL | LPAREN Type RPAREN
Type ::= BaseType | BaseType ARROW Type

ExpSeq ::= Exp | Exp COMMA ExpSeq

Var ::= VAR(s)

AddOp ::= PLUS | MINUS
MulOp ::= TIMES
RelOp ::= EQUALS | LESSTHAN
UnaryOp ::= NEGATE

FactorA ::= LPAREN Exp RPAREN
      | NUMBER(n)
      | Var
      | TRUE
      | FALSE
      | IF Exp THEN Exp ELSE Exp FI
      | LET Var EQUALS Exp IN Exp END
      | FN Var COLON Type DARROW Exp
      | REC Var COLON Type DARROW Exp
      | FUN Var LPAREN Var COLON Type RPAREN COLON Type DARROW Exp
      | UnaryOp Factor

Factor ::= FactorA
      | Factor Exp

Term ::= Factor
      | Factor MulOp Term

Exp’ ::= Term
     | Term AddOp Exp

Exp ::= Exp’
     | Exp’ RelOp Exp

Program ::= Exp SEMICOLON
```

Figure 1: MinML concrete syntax.

Concrete Syntax	**Lexer Tokens**	**Abstract Syntax**
`true`	`TRUE`	`Bool(true)`
`1`	`NUMBER(1)`	`Int(1)`
`1 + 2`	`NUMBER(1) PLUS NUMBER(2)`	`Primop(Plus, [Int(1), Int(2)])`
`if true then 4 else 5 fi`	`IF TRUE THEN NUMBER(4) ELSE NUMBER(5) FI`	`If(Bool(true), Int(4), Int(5))`
`f 3`	`VAR("f") NUMBER(3)`	`Apply(Var("f"), Int(3))`
`fn g : int => true`	`FN VAR("g") COLON INT DARROW TRUE`	`Fn("g", INT, Bool(true))`
`rec f : int -> bool => fn g : int => true`	`REC VAR("f") COLON INT ARROW BOOL DARROW FN VAR("g") COLON INT DARROW TRUE`	`Rec("f", ARROW(INT,BOOL), Fn("g", INT, Bool(true)))`
`fun f (g : int) :  bool => true`	`FUN VAR("f") LPAREN VAR("g") COLON INT RPAREN COLON BOOL DARROW TRUE`	`Rec("f", ARROW(INT,BOOL), Fn("g", INT, Bool(true)))`

Figure 2: Examples of MinML syntax.

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x : \tau}\ \textit{VarTyp} \qquad \frac{}{\Gamma \vdash \mathtt{num}(n) : \mathtt{int}}\ \textit{NumTyp}$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}}\ \textit{TrueTyp} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}\ \textit{FalseTyp}$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}(e, e_1, e_2) : \tau}\ \textit{IfTyp} \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \mathtt{rec}(\tau, x.e) : \tau}\ \textit{RecTyp}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fn}(\tau_1, x.e) : \tau_1 \to \tau_2}\ \textit{FnTyp} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) : \tau}\ \textit{AppTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_{o1} \quad \ldots \quad \Gamma \vdash e_n : \tau_{on}}{\Gamma \vdash o(e_1, \ldots, e_n) : \tau_o}\ \textit{OpTyp} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}(e_1, x.e_2) : \tau_2}\ \textit{LetTyp}$$

Figure 3: Static semantics for MinML.

which has variables not bound by a `let`, `rec`, and `fn`. In the translator, you will need to maintain an environment of variable names. Use the simplest representation possible; don't worry about efficiency.

1.2 Typechecker

Next, you will implement a typechecker for MinML. The static semantics for MinML, given in Figure 3, assures that every expression has at most one type. Therefore, your typechecker will return the unique type for an expression if it is well-typed or raise the exception `Typing.Error` otherwise.

Recall that the specification of MinML uses a *typing judgment* to classify MinML expressions as ill- or well-typed. The typing judgment is defined inductively by a set of inference rules. It follows that, in order to decide whether a given expression has a type, we need to search for a derivation using the typing rules. A moment of thought reveals that if we could classify an expression as ill- or well-typed through a typing judgment, then we could also retrieve its type easily, because the derivation itself would us tell exactly how to determine the type of the expression. So, in fact, deciding the type of an expression is no harder than deciding if the expression is ill- or well-typed.

In general, we cannot assume that an expression matches only one typing rule, thus the search strategy for a derivation can be *non-deterministic*. Fortunately, the search strategy for MinML is *syntax directed*: the form of expression we are typing determines uniquely which rule to apply. Therefore, if the typechecker finds that no rule can be applied to an expression, it knows that the expression is ill-typed and can raise an exception immediately without backtracking. Your code will probably have one function clause for each constructor of the datatype `DBMinML.exp`.

We provide a function `MinML.typeOfPrimop` that returns the domain and range types for a primitive operation. Your typechecker should use this function, but should not rely on the fact that all primitive operations currently have a maximum of two arguments; it should be possible to add new operations to MinML without modifying your type checker.

Task: Typechecker (35 points)
Complete the code in `typing.sml` to produce a structure `Typing :> TYPING` which imple-

$$\frac{e_i \mapsto e_i'}{o(v_1, \ldots, e_i, \ldots, e_n) \mapsto o(v_1, \ldots, e_i', \ldots, e_n)}\ \textit{OpArg} \qquad \frac{\text{(by primop } o)}{o(v_1, \ldots, v_n) \mapsto v}\ \textit{OpVals}$$

$$\frac{e \mapsto e'}{\texttt{if}(e, e_1, e_2) \mapsto \texttt{if}(e', e_1, e_2)}\ \textit{IfCond}$$

$$\frac{}{\texttt{if}(\texttt{true}, e_1, e_2) \mapsto e_1}\ \textit{IfTrue} \qquad \frac{}{\texttt{if}(\texttt{false}, e_1, e_2) \mapsto e_2}\ \textit{IfFalse}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)}\ \textit{AppFun} \qquad \frac{e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')}\ \textit{AppArg}$$

$$\frac{}{\texttt{apply}(\texttt{fn}(\tau, x.e), v_2) \mapsto \{v_2/x\}e}\ \textit{CallFn} \qquad \frac{}{\texttt{rec}(\tau, x.e) \mapsto \{\texttt{rec}(\tau, x.e)/x\}e}\ \textit{UnfoldRec}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1, x.e_2) \mapsto \texttt{let}(e_1', x.e_2)}\ \textit{LetArg} \qquad \frac{}{\texttt{let}(v_1, x.e_2) \mapsto \{v_1/x\}e_2}\ \textit{Let}$$

Figure 4: Dynamic semantics for MinML (v, v_i, etc. denote expressions that are values).

ments the behavior specified. You should not modify any other files. Remember that the expression to be typechecked will be in deBruijn form. This file contains some code to get you started; we recommend using it. We also recommend that you write a function called `typing`, with type `(typ env) * exp -> typ` and specification as follows:

> Given a type environment Γ and an expression e, `typing` returns τ, the type of e under Γ, if τ exists. If e is ill-typed in Γ then `typing` raises the exception `Typing.Error`.

You can test your typechecker before you complete the evaluator. Run `Top.loop_type ();` or `Top.file_type "test_file.mml";`.

1.3 Evaluator

Finally, you'll implement the MinML dynamic semantics in the file `eval.sml`. The dynamic semantics is given in Figure 4 as a relation "$\mapsto$" for single-step evaluation. There are many more efficient ways of evaluating MinML programs (as we'll see later in the class), but we require that you strictly follow the specified semantics for this assignment.

The general idea of this style of specifying the evaluation is that one continues to take steps until reaching a value, or getting 'stuck'. A value in this language is defined as an expression which is one of the following forms: `Int`(i), `Bool`(b), or `Lam`(e). That is, a literal integer, a literal boolean, or a function expression. Being 'stuck' is defined as considering an expression which is *not* a value, but not being able to take a step.

The evaluation algorithm is straightforward. First it will use the "search rules" *OpArg*, *IfCond*, *AppFun*, *AppArg*, and *LetArg* to recursively scan the input expression for the proper subexpression to modify. Once the proper subexpression has been located, one of the "instruction rules" *OpVals*, *IfTrue*, *IfFalse*, *CallFun*, *UnfoldRec*, *Let* can be applied. If no rule applies (as might happen if the expression is already fully evaluated, or is ill-typed), the evaluator will raise an exception.

For example, on the expression $e_1\ e_2$, the evaluator will try to apply an evaluation step to the function expression e_1. If it is already a value the evaluator will try to apply a step to the argument expression e_2. If it is already a value as well, the evaluator will try to use the instruction rule for application, *CallFun*.

Since the *CallFun*, *UnfoldRec*, and *Let* rules involve substitutions, you will also need to properly implement substitutions. This isn't hard, but as usual, think before you code.

Task: Evaluator (45 points)
In `eval.sml`, fill in the structure `Eval :> EVAL` to implement the behavior specified. You should not modify any other files. Most of the work that you do will be in the function `step`, which has type `exp -> exp` and the specification:

> Given an expression e in deBruijn form, `step` returns the unique e' such that $e \mapsto e'$. If no such e' exists, `step` raises the exception `Eval.Stuck` if e is not a value, or `Eval.Done` if it is.

Test Cases

We have provided a small number of test cases. Once your evaluator is complete, these may be run by typing `Top.file_eval "test_file.mml";`. These test cases are described in the table below.

Filename	Expected Result	Description
`if.mml`	`3 : int`	Simple test of `if`
`fun.mml`	`fn _ :int => DB[1] : (int) -> (int)`	Simple test of `fun`
`factorial.mml`	`120 : int`	The factorial function
`self.mml`	*ill-typed*	Ill-typed function
`hof.mml`	`7 : int`	Simulates pairs using functions

These test files are (obviously) not exhaustive, so you should develop your own in order to test your program thoroughly. You are encouraged to submit test cases to us. We will test each solution against a subset of the submitted test cases, in addition to our own. So, even though you will not receive any points for handing in test cases, it is in your interest to send us tests that your code handles correctly: it will tend to improve your grade. See below for submission instructions.

2 Hand-in Instructions

Turn in the three files `translate.sml`, `typing.sml`, and `eval.sml` by copying them to your handin directory

`/afs/andrew/scs/cs/15-312/students/`*Andrew user ID*`/asst2/`

by 11:59 PM on the due date. Immediately after the deadline, we will run a script to sweep through all the handin directories and copy your files elsewhere. We will also sweep 24, 48, and 72 hours after the deadline, for anyone using late days on this assignment.

Also, please turn in any test cases you'd like us to use by copying them to your handin directory. To ensure that our scripts notice these files, make sure they have the suffix `.mml`.

For more information on handing in code, refer to

```
http://www.cs.cmu.edu/~fp/courses/312/assignments.html
```