Assignment 2:
Implementing MinML

15-312: Foundations of Programming Languages
Joshua Dunfield (joshuad@cs.cmu.edu)

Out: Thursday, September 5, 2002
Due: Thursday, September 19, 2002 (11:59 pm)

100 points total

Revised September 10, 2002

This revised version corrects some typos and inaccuracies in the original handout. For a summary of changes, refer to

http://www.cs.cmu.edu/~fp/courses/312/assignments/asst2/index.html

1 MinML

For this assignment you will implement a typechecker and evaluator for MinML. In the assignment folder you'll 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. Hence, you should strive for elegance.

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 ParseError with a (somewhat) informative message.

While we've 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 `INT`, `BOOL`, etc. The tokens are defined in Figure 2; the lexer 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

```
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
      | FUN Var LPAREN Var COLON Type RPAREN COLON Type IS Exp END
      | 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.

Symbol	**Lexer.token**
`int`	`INT`
`bool`	`BOOL`
`->`	`ARROW`
`true`	`TRUE`
`false`	`FALSE`
`fun`	`FUN`
`is`	`IS`
`end`	`END`
`if`	`IF`
`then`	`THEN`
`else`	`ELSE`
`fi`	`FI`
`let`	`LET`
`in`	`IN`
`,`	`COMMA`
`(`	`LPAREN`
`)`	`RPAREN`
`;`	`SEMICOLON`
`~`	`NEGATE`
`=`	`EQUALS`
`<`	`LESSTHAN`
`*`	`TIMES`
`-`	`MINUS`
`+`	`PLUS`
`:`	`COLON`
n	`NUMBER(n)`
any other string *s*	`VAR(s)`

Figure 2: MinML tokens.

multiplication, then addition and subtraction, and finally equality). All primitive operations are left-associative. (The grammar is actually *right*-associative, since that's easier to implement in a recursive-descent style, so we have 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`.)

The type constructor '`->`' is infix and right-associative, just as in SML.

Here are some examples along with their translation into MinML abstract syntax (type `MinML.exp`).

Concrete Syntax	**Lexer Tokens**	**Abstract Syntax**
`true`	`TRUE`	`Bool(true)`
`1`	`NUMBER(1)`	`Int(1)`
`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))`
`1 + 2`	`NUMBER(1) PLUS NUMBER(2)`	`Primop(Plus, [Int(1), Int(2)])`
`1 + g 2 * 3`	`NUMBER(1) PLUS Var("g") NUMBER(2) TIMES NUMBER(3)`	`Primop(Plus, [Int(1), Primop(Times, [Apply(Var("g"), 2), Int(3)])])`
`fun f(g : int -> bool -> int) : bool is true end`	`FUN VAR("f") LPAREN VAR("g") COLON INT ARROW BOOL ARROW INT RPAREN COLON BOOL IS TRUE END`	`Fun(ARROW(INT,ARROW(BOOL,INT)), BOOL, ("f", "g", Bool(true)))`

The abstract syntax groups binders with their scope, in the style of higher-order abstract syntax. However, variables are represented via their name as a string.

To play around with the parser and become familiar with MinML, 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 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`, and `fun` require a little thought. For `fun`, which binds *two* variables at once (the function and its argument), use the convention that the de Bruijn index `1` refers to the argument and the de Bruijn index `2` to the function itself. For example:

Named	**deBruijn**

`fun fact (x : int) : int is`
`if x=0 then 1 else x*fact(x-1)`
`fi`
`end`
	`fun _ (_ : int) : int is`
	`if 1=0 then 1 else 1*2(1-1) fi`
	`end`

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, 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 `TypeError` 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 the inference rules. Therefore, in order to decide whether a given expression has a type, we need to search for a derivation using the typing rules. However, a moment of thought realizes 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 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 and 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 primop. Your typechecker should use this function, but should not rely on the fact that the primops currently have a maximum of two arguments; it should be possible to add primops to MinML without modifying your type checker.

Task: Typechecker (35 points)
Complete the code in `typing.sml` to produce a structure `Typing :> TYPING` which implements 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 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 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*, *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* 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 `Stuck`.

Test Cases

We've provided a few test cases. Once your evaluator is complete, these may be run in the following manner:

```
Top.file_eval "test_file.mml";
```

Filename	Expected Result	Description
`if.mml`	`3 : int`	Simple test of `if`
`fun.mml`	`fun ident(x : int) : int is x end : 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. Remember, however, that passing test cases is only a *necessary* condition for getting a good grade on your homework. The elegance of your solution is important and will be taken into account when grading.

You are encouraged to submit test cases to us. We will test everyone's code against a subset of the submitted test cases, in addition to our own. So, even though you will not receive any points specifically for handing in test cases, it's 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 the 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`

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x : \tau}\ \textit{VarTyp}$$

$$\frac{}{\Gamma \vdash \texttt{num}(n) : \texttt{int}}\ \textit{NumTyp}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textit{TrueTyp} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}(e, e_1, e_2) : \tau}\ \textit{IfTyp}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textit{FalseTyp}$$

$$\frac{\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}(\tau_1, \tau_2, f.x.e) : \tau_1 \to \tau_2}\ \textit{FunTyp} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau}\ \textit{AppTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_{o1} \quad \ldots \quad \Gamma \vdash e_n : \tau_{on}}{\Gamma \vdash o(e_1, \ldots, e_n) : \tau_o}\ \textit{OpTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1, x.e_2) : \tau_2}\ \textit{LetTyp}$$

Figure 3: Static semantics for MinML

$$\frac{e_i \mapsto e_i'}{o(v_1, \ldots, e_i, \ldots, e_n) \mapsto o(v_1, \ldots, e_i', \ldots, e_n)}\ \textit{OpArg}$$

$$\frac{\text{(by primop } o)}{o(v_1, \ldots, v_n) \mapsto v}\ \textit{OpVals}$$

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

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

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

$$\frac{e_1 \mapsto e_1'}{\texttt{Apply}(e_1, e_2) \mapsto \texttt{Apply}(e_1', e_2)}\ \textit{AppFun}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{Apply}(v_1, e_2) \mapsto \texttt{Apply}(v_1, e_2')}\ \textit{AppArg}$$

$$\frac{v_2 = \texttt{fun}(\tau_1, \tau_2, f.x.e)}{\texttt{Apply}(v_2, v_1) \mapsto \{v_2/f\}\{v_1/x\}e}\ \textit{CallFun}$$

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

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

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