Assignment 4:
Evaluation in the E Machine

15-312: Foundations of Programming Languages
Daniel Spoonhower (spoons@cs.cmu.edu)
Edited by Jason Reed (jcreed@cs.cmu.edu)

Out: Thursday, September 30, 2004
Due: Thursday, October 14, 2004 (11:59 pm)

100 points total + 40 points extra credit

1 Introduction

To begin this assignment, you will first extend the parser from Assignment 2. You will then implement an E machine evaluator (as shown below) that supports unit, pairs, functions, recursive expressions, and named exceptions.

In the assignment directory you'll find several files with support code; you will only need to fill in the missing code in `parse.sml`, `typing.sml`, and `e-mach.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 (re)read 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()`.

Note: In the `.sml` files, significant changes from Assignment 2 are indicated as shown below.

```
(* new asst4 code: *)
...
(* end asst4 code *)
```

2 The E Machine

For this assignment we will be creating closures for functions instead of performing substitutions. The E machine is an abstract machine like the C machine, except that its stack may contain *value environments* η as well as *control frames* f. The scope of such a value environment is every control frame and expression to its right.

States	s	$::=$	$k > e$	evaluate e under k
		$\mid$	$k < v$	return v to k
Stacks	k	$::=$	$\bullet$	empty stack
		$\mid$	$k \triangleright f$	stack k plus control frame f
		$\mid$	$k \blacktriangleright \eta$	stack k plus value environment η
Frames	f	$::=$	$\texttt{o}(\square, e_2) \mid \texttt{o}(v_1, \square)$	primops
		$\mid$	$\texttt{pair}(\square, e_2) \mid \texttt{pair}(v_1, \square)$	pairs
		$\mid$	$\texttt{fst}(\square) \mid \texttt{snd}(\square)$	projections
		$\mid$	$\texttt{apply}(\square, e_2) \mid \texttt{apply}(v_1, \square)$	applications
		$\mid$	$\texttt{if}(\square, e_1, e_2)$	conditional

We often need to find out the most current (i.e. most recent) value environment of a stack. We denote this operation by $\eta(k)$. This is defined recursively by

$$\eta(\bullet) = \cdot$$

$$\eta(k \triangleright f) = \eta(k)$$

$$\eta(k \blacktriangleright \eta) = \eta$$

3 Parser and Concrete Syntax

The concrete syntax for this assignment is shown in Figure 1 (though we will continue to use abstract syntax for types to improve readability). The grammar differs from Assignment 2 as follows:

- MinML now includes unit and pair (product) types $\tau_1 * \tau_2$. The type constructor $*$ has higher precedence than $\rightarrow$, just as in SML. Unlike SML, however, there are only pairs, not arbitrary n-tuples. For example, `int` $*$ `int` $*$ `int` is not syntactically correct; you have to write either (`int` $*$ `int`) $*$ `int` or `int` $*$ (`int` $*$ `int`). (Since there's no obvious associativity, we require parentheses.)
- We have also added `exn`, the type of exceptions.
- The remaining changes are all to the syntactic category `FactorA`. We will show examples of the new constructs below.

Finally, the lexer now supports SML-style comments; you may now annotate your test programs with text that will be ignored by the typechecker and evaluator.

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

```
(* new or changed in asst4: *)
BaseType ::= INT | BOOL | UNIT | EXN | LPAREN Type RPAREN
PairType ::= BaseType | BaseType TIMES BaseType
Type ::= BaseType | PairType ARROW Type
(* end asst4 *)
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
(* new in asst4: *)
    (* Unit element *)
      | LPAREN RPAREN
    (* Pairs *)
      | LPAREN Exp COMMA Exp RPAREN
      | FST FactorA
      | SND FactorA
    (* Exceptions *)
      | EXCEPTION Var IN Exp END
      | RAISE LBRACKET Type RBRACKET FactorA
      | TRY Exp CATCH Exp WITH Exp END
(* end asst4 *)
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
`(1,2)`	`LPAREN NUMBER(1) COMMA NUMBER(2) RPAREN`	`Pair(Int(1), Int(2))`
`fst x`	`FST VAR("x")`	`Fst(Var("x"))`
`exception x in () end`	`EXCEPTION VAR("x") IN LPAREN RPAREN END`	`Exception(Var("x"), Unit)`
`raise[int*int] ex`	`RAISE LBRACKET INT TIMES INT RBRACKET VAR("ex")`	`Raise(PAIR(INT,INT), Var("ex"))`

Just as in Assignment 2, abstract syntax groups binders with their scope, in the style of higher-order abstract syntax, and variables are represented via their name as a string.

In this assignment, MinML already supports a limited form of evaluation, in particular, the evaluation of arithmetic expressions. You may experiment with the current parser, typechecker and evaluator by typing `Top.loop_eval ();` or `Top.file_eval "test_file.mml";` at the SML/NJ prompt.

Task: Parsing (10 points)
Extend the implementation in `parse.sml` to handle all the new expression forms: unit, pairs, `fst`, `snd`, `exception`, `raise`, and `try`. **Hint:** Focus your attention on `parse_factora`. The new type constructors $*$ and `exn` have been implemented for you.

4 De Bruijn Translation

De Bruijn translation of the new constructs has been implemented for you. Except for the parser (which emits an abstract syntax tree in named form), all of your code will operate on programs in de Bruijn form.

5 Unit and Pairs

In this task you will add unit and pairs to MinML. The typing rules shown below should be familiar. Unlike Assignment 2, the dynamic semantics below describes *eager* pairs; $\texttt{pair}(v_1, v_2)$ is a value only if both v_1 and v_2 are values.

Notice the four different forms of frames that appear in the evaluation rules below. You will find four corresponding constructors `FPair1`, `FPair2`, `FFst`, and `FSnd` in the `frame` datatype in `e-mach.sml`. These constructors have already been given the proper parameter types for you to use in your implementation.

$$\frac{}{\Gamma \vdash \texttt{unitel} : \texttt{unit}} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair}(e_1, e_2) : \tau_1 * \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \texttt{fst}(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \texttt{snd}(e) : \tau_2}$$

$$\begin{array}{lcl}
k > \texttt{unitel} & \mapsto_{\mathsf{e}} & k < \texttt{unitel} \\
\\
k > \texttt{pair}(e_1, e_2) & \mapsto_{\mathsf{e}} & k \triangleright \texttt{pair}(\square, e_2) > e_1 \\
k \triangleright \texttt{pair}(\square, e_2) < v_1 & \mapsto_{\mathsf{e}} & k \triangleright \texttt{pair}(v_1, \square) > e_2 \\
k \triangleright \texttt{pair}(v_1, \square) < v_2 & \mapsto_{\mathsf{e}} & k < \texttt{pair}(v_1, v_2) \\
\\
k > \texttt{fst}(e) & \mapsto_{\mathsf{e}} & k \triangleright \texttt{fst}(\square) > e \\
k \triangleright \texttt{fst}(\square) < \texttt{pair}(v_1, v_2) & \mapsto_{\mathsf{e}} & k < v_1 \\
\\
k > \texttt{snd}(e) & \mapsto_{\mathsf{e}} & k \triangleright \texttt{snd}(\square) > e \\
k \triangleright \texttt{snd}(\square) < \texttt{pair}(v_1, v_2) & \mapsto_{\mathsf{e}} & k < v_2
\end{array}$$

Task: Unit and Pairs: Typing and Evaluation (15 points)
Following the rules above, extend (1) the `typing` function in `typing.sml` and (2) the `step` function in `e-mach.sml` to handle unit and pairs (including `fst` and `snd`). You may start with your own solution to Assignment 2 or the version of `typing.sml` we have provided.

Note that at this stage of your implementation, your evaluator will remain limited in the scope of programs it can handle. Use the example file `pairs.mml` (or equivalent) and the function `Top.file_step` to see the intermediate steps taken by your evaluator.

6 Closures and Suspensions

The E machine has no intrinsic notion of substitution; instead, it explicitly maintains a mapping from variable bindings to the expressions they stand for. In the rules below, we assumed that each variable is bound at most once in every environment. Your implementation, however, need not be concerned with the uniqueness of bindings, since it will operate on expressions in de Bruijn form.

Remember that in the semantics of the E machine, function expressions are *not* values. Instead, we bind the most recent environment together with an expression $\texttt{fn}(\tau, x.e)$ to form a closure. Similarly, you should also include copy of the most recent environment when binding recursive expressions and let-expressions into the environment.

For this part of the assignment, you should bind the entire most recent environment into each closure as shown in the rules below.

$$
\begin{array}{lll}
k > \texttt{fn}(\tau, x.e) & \mapsto_{\texttt{e}} & k < \langle\!\langle \eta(k); \texttt{fn}(\tau, x.e) \rangle\!\rangle \\
k > \texttt{apply(}e_1, e_2\texttt{)} & \mapsto_{\texttt{e}} & k \triangleright \texttt{apply(}\square, e_2\texttt{)} > e_1 \\
k \triangleright \texttt{apply(}\square, e_2\texttt{)} < v_1 & \mapsto_{\texttt{e}} & k \triangleright \texttt{apply(}v_1, \square\texttt{)} > e_2 \\
k \triangleright \texttt{apply(}\langle\!\langle \eta; \texttt{fn}(\tau, x.e) \rangle\!\rangle, \square\texttt{)} < v_2 & \mapsto_{\texttt{e}} & k \blacktriangleright (\eta, x = v_2) > e \\
 & & \\
k > \texttt{let}(e_1, x.e_2) & \mapsto_{\texttt{e}} & k \triangleright \texttt{let}(\square, x.e_2) > e_1 \\
k \triangleright \texttt{let}(\square, x.e_2) < v_1 & \mapsto_{\texttt{e}} & k \blacktriangleright (\eta(k), x = v_1) > e_2 \\
k > x & \mapsto_{\texttt{e}} & k < v \\
 & & \eta(k) = (\ldots, x = v, \ldots) \\
k \blacktriangleright \eta < v & \mapsto_{\texttt{e}} & k < v \\
 & & \\
k > \texttt{rec}(\tau, x.e) & \mapsto_{\texttt{e}} & k \blacktriangleright (\eta(k), x \stackrel{*}{=} \langle\!\langle \eta(k); \texttt{rec}(\tau, x.e) \rangle\!\rangle) > e \\
k > x & \mapsto_{\texttt{e}} & k \blacktriangleright \eta' > e \\
 & & \eta(k) = (\ldots, x \stackrel{*}{=} \langle\!\langle \eta'; e \rangle\!\rangle, \ldots)
\end{array}
$$

The notation $\stackrel{*}{=}$ denotes an entry in the environment that binds a mere *expression* to a variable, which is not necessarily a value. This is necessary for the bindings that arise from `rec`-expressions. In your code use `VSuspend` to create this special kind of closure, and `VClosure` for function closures.

Task: Closures and Suspensions: Evaluation (25 points)
Following the rules above, (1) modify the `frame` and `value` datatypes in `e-mach.sml` as necessary and (2) extend the `step` function to handle `fn`, `let`, `rec`, and `apply`.

Note that the functions `Top.loop_eval` and `Top.file_eval` will *not* print the result of a program that evaluates to an function expression $\texttt{fn}(\tau, x.e)$ as they are currently implemented. You may modify the implementation to do so or use `Top.loop_step` and `Top.file_step` to see the result of such an evaluation.

Task: Efficient Closures: Evaluation (EXTRA CREDIT, 15 points)
The rules above bind *all* variables found in the current environment into a closure. Type safety requires only those variables that are free in the enclosed expression to be bound in the closure. Modify your implementation so that the minimum number of variables are bound each closure. (Think carefully about about how our use of de Bruijn indices would affect this optimization.) If you undertake this task, you may hand in a separate file `smart-e-mach.sml` rather than modifying your previous implementation.

7 Named Exceptions

$$
\frac{\Gamma \vdash e : \mathsf{exn}}{\Gamma \vdash \texttt{raise}(\tau, e) : \tau} \qquad \frac{\Gamma, x{:}\mathsf{exn} \vdash e : \tau}{\Gamma \vdash \texttt{exception}(x.e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathsf{exn} \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{try}(e_1, e_2, e_3) : \tau}
$$

Task: Named Exceptions: Typing (5 points)
Following the rules above, extend the `typing` function in `typing.sml` to handle the `exception`, `raise`, and `try` constructs.

Task: Named Exceptions: Transition Rules (15 points)
Formulate all the `E` machine transition rules needed for `exception`, `raise`, and `try`. Use a notation like

$$k \ll \varepsilon$$

to denote a state in which the exception value ε has been raised with stack k. Exception values ε are a subset of values v. Your rules (and your implementation) *should* allow exceptions to escape the scope in which they were created. Furthermore, you should describe an evaluation in which the second expression e_2 of `try` is *always* evaluated (and therefore e_2 should be evaluated before e_1.)

Here are some examples of what behavior we expect:

- `exception`$(x.$`try`$(5, x, 6))$ evaluates to 5.
- `exception`$(x.$`try`$(7 +$ `raise`$(x), x, 6))$ evaluates to 6.
- `exception`$(x.$`exception`$(y.$`try`$(7 +$ `raise`$(y), x, 6)))$ raises an exception at the top level.
- `exception`$(x.$`exception`$(y.$`try`$(7 +$ `raise`$(x),$ `fst`$(x,$ `raise`$(y)), 6)))$ raises an exception at the top level.
- `exception`$(x.$`exception`$(y.$`try`$($`try`$($`try`$($`raise`$(x),$ `raise`$(y), 0), x, 7), y, 8))$ evaluates to 8.
- `exception`$(x.$`exception`$(y.$`try`$($`try`$($`try`$($`raise`$(x), y, 0), x, 7), y, 8))$ evaluates to 7.
- `exception`$(x.$`exception`$(y.$`try`$($`try`$($`try`$($`raise`$(y), y, 0), x, 7), y, 8))$ evaluates to 0.
- `exception`$(x.$`exception`$(y.$`try`$($`try`$($`raise`$(x),$ `raise`$(y), 0), x, 7))$ raises an exception at the top level.

Submit the rules on paper or as a file `exn-rules.txt` or `exn-rules.ps`. If you want, you can email your rules to me (`jcreedandrew.cmu.edu`) before you start on implementation and I will let you know if you're on the right track so you don't spend a lot of time implementing something incorrect.

Task: Named Exceptions: Implementation (30 points)
Implement your rules in `e-mach.sml`. You will need to figure out a suitable representation for exceptions and change the definition of the `VExn` constructor accordingly.

Although it is absolutely correct to treat `exception x in e end` as a binding occurrence of x, and to use a deBruijn index for the vairable x when in occurs in e, it **will not work** to simply copy this deBruijn index as a run-time representation of the 'live' exception value. This was a common mistake in previous years' solutions to this assignment. The important properties to maintain are that `exception x in e end` should generate a fresh exception when it is executed, which may escape its scope, and that the evaluator must be able to accurately recognize when two exceptions are equal to one another, for the `try` construct to work correctly.

8 Lazy Evaluation [Extra Credit]

Following your work in Assignment 2, in this extra credit task, you will add a construct for lazy evaluation to MinML. The concrete syntax should be of the form `delay` e, and we will use the abstract syntax `delay`(e). The semantics of `delay`(e) is that the evaluation of e is suspended, and a suspension is returned as a value. The suspension is opened and evaluated as late as possible. For example, a suspension of pair type should only be opened when it appears as an argument to `fst` or `snd`; a suspension of function type should only be opened when it appears on the left side of an application. Similarly for `try` and `raise`.

Task: Lazy Evaluation: Parsing, Typing and Evaluation (EXTRA CREDIT, 25 points) Extend the lexer and parser to accept this new construct. Write down the static and dynamic semantics and implement them in `typing.sml` and `e-mach.sml`. Hand in your typing and evaluation rules on paper or as a file `lazy-rules.txt` or `lazy-rules.ps`. Partial credit will be given if you correctly complete one or more of these subtasks.

9 Test Cases

Filename	Expected Result	Description

`pairs.mml`	`((3, (true, 5)), ())`
	`  : ((int) * ((bool)`
	`  * (int))) * (unit)`
		Simple test of `pair` and `unit`

`primops.mml`	`true : bool`	Simple test of arithmetic
`uncaught.mml`	raises `Uncaught`	Raises an uncaught exception

You are encouraged to submit test cases to us. We will test each submission 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. See below for submission instructions.

10 Hand-in Instructions

Turn in the files `parse.sml`, `typing.sml`, and `e-mach.sml` (and any additional files you modify in the extra credit tasks) by copying them to your handin directory

`/afs/andrew/scs/cs/15-312/students/`*Andrew user ID*`/asst4/`

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.

Turn in non-programming questions as text or postscript files in the handin directory. Or, if you wish, you may turn in answers on paper, due in WeH 3721 by 11:59 pm on the due date. **If you are using late days, paper handin is by arrangement only** (send mail and we'll figure something out).

Also, please turn in any test cases you'd like us to use by copying them to your handin directory. To ensure that we 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`