Assignment 4:
Control Flow in the C-machine

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

Out: Thursday, September 26, 2002
Due: Thursday, October 10, 2002 (11:59 pm)

100 points total + 25 points extra credit

1 Introduction

In this assignment, you will extend the parser from Assignment 2, and the continuation-based C-machine evaluator shown in lecture, to support unit, pairs, first-class continuations, 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 `c-mach.sml`. (For named exceptions, you will also need to change two lines in `minml.sml`.) You will rarely, if ever, need to write long or complicated functions to complete this assignment. Hence, you should strive for elegance. You may wish to (re)read through the provided code 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, most changes from Assignment 2 are indicated like this:

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

2 Parser and Concrete Syntax

The concrete syntax for this assignment is shown in Figure 1, and the tokens are listed in Figure 2. The grammar differs from Assignment 2 as follows:

- There are now unit and pair (product) types τ_1`*`τ_2. The type constructor `*` has higher precedence than `->`, just as in SML. Unlike SML, 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.)
- There are also types τ`cont` (for any type τ) and `exn`, the type of exceptions.

- The remaining changes are all to `FactorA`. We'll show examples of the new constructs later in the assignment.

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

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"))`
`letcc[unit] k in 0 end`	`LETCC LBRACKET UNITTYPE RBRACKET VAR("k") IN NUMBER(0) END`	`Letcc(UNIT, ("k", Int(0)))`
`throw[int] x to k`	`THROW LBRACKET INT RBRACKET VAR("x") TO VAR("k")`	`Throw(INT, Var("x"), Var("k"))`
`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.

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: Parsing (15 points)
Extend the implementation in `parse.sml` to handle all the new expression forms: unit, pairs, `fst`, `snd`, `letcc`, `throw`, `exception`, `raise`, and `try`. **Hint:** Focus your attention on `parse_factora`. The new type constructors `*`, `cont`, and `exn` have been implemented for you.

3 DeBruijn Translation

DeBruijn 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 with programs in deBruijn form.

```
(* new or changed in asst4: *)
BaseType ::= INT | BOOL | UNIT | EXN | LPAREN Type RPAREN
ContType ::= BaseType | BaseType CONT
PairType ::= ContType | ContType TIMES ContType
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
      | FUN Var LPAREN Var COLON Type RPAREN COLON Type IS Exp END
      | UnaryOp Factor
(* new in asst4: *)
    (* Unit value *)
      | LPAREN RPAREN
    (* Pairs *)
      | LPAREN Exp COMMA Exp RPAREN
      | FST FactorA
      | SND FactorA
    (* First-class continuations *)
      | LETCC LBRACKET Type RBRACKET Var IN Exp END
      | THROW LBRACKET Type RBRACKET Exp TO Exp
    (* Exceptions *)
      | EXCEPTION Var IN Exp END
      | RAISE LBRACKET Type RBRACKET FactorA
      | TRY Exp CATCH Exp WITH Exp END
(* end *)
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)`
new in asst4:	
`[`	`LBRACKET`
`]`	`RBRACKET`
`fst`	`FST`
`snd`	`SND`
`letcc`	`LETCC`
`throw`	`THROW`
`to`	`TO`
`exception`	`EXCEPTION`
`try`	`TRY`
`catch`	`CATCH`
`with`	`WITH`
`raise`	`RAISE`
end asst4	
any other string *s*	`VAR(s)`

Figure 2: MinML tokens.

4 Unit and Pairs

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

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

Task: Unit and Pairs: Typing and Evaluation (15 points)
Following the rules above, extend (1) the `typing` function in `typing.sml` to and (2) the `eval` function in `c-mach.sml` to handle unit and pairs.

5 First-Class Continuations

Let's add first-class continuations to MinML. The type of a continuation that expects a value of type τ is τ `cont`; the `letcc` construct binds the current continuation and the `throw` construct passes a value to a continuation. Refer to the lecture notes and Chapter 12 of Harper's book for thorough coverage of these constructs.

$$\frac{\Gamma, x{:}\tau\,\texttt{cont} \vdash e : \tau}{\Gamma \vdash \texttt{letcc}(\tau, x.e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau'\,\texttt{cont}}{\Gamma \vdash \texttt{throw}(\tau, e_1, e_2) : \tau}$$

$$\begin{array}{lcl}
k > \texttt{letcc}(\tau, x.e) & \mapsto_{\texttt{c}} & k > \{\texttt{cont}(k)/x\}e \\
k > \texttt{throw}(\tau, e_1, e_2) & \mapsto_{\texttt{c}} & k \triangleright \texttt{throw}(e_1, e_2) > e_1 \\
k \triangleright \texttt{throw}(e_1, e_2) < v_1 & \mapsto_{\texttt{c}} & k \triangleright \texttt{throw}(v_1, \square) > e_2 \\
k \triangleright \texttt{throw}(v_1, \square) < \texttt{cont}(k') & \mapsto_{\texttt{c}} & k' < v_1
\end{array}$$

A continuation k is represented by the MinML constructor `Cont : exp -> exp`. Note that such expressions have no concrete syntax; they only arise during evaluation. The type-checker which is part of the front end must therefore disallow them. This means that there will be no way to typecheck some intermediate machine state in your implementation.

Task: First-Class Continuations: Typing and Evaluation (20 points)
Following the rules above, extend (1) the `typing` function in `typing.sml` to and (2) the `eval` function in `c-mach.sml` to handle `letcc` and `throw`.

6 Named Exceptions

$$\frac{\Gamma \vdash v : \mathsf{exn} \quad v\ \mathsf{value}}{\Gamma \vdash \texttt{raise}(\tau, v) : \tau} \qquad \frac{\Gamma, x{:}\mathsf{exn} \vdash e : \tau}{\Gamma \vdash \texttt{exception}(x.e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash v_2 : \mathsf{exn} \quad v_2\ \mathsf{value} \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{try}(e_1, v_2, e_3) : \tau}$$

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. Note that inside a program, a variable always stands for a value, so x value holds for any variable x.

Task: Named Exceptions: Transition Rules (15 points)
Formulate all the C-machine transition rules needed for `exception`, `raise`, and `try`. As in lecture, use the notation

$$k \ll v$$

to denote a state in which the exception v has been raised with stack k. Your rules (and your implementation) *should* allow exceptions to escape (non-escaping exceptions are extra credit; see below). Moreover, your rules need to work correctly *only* in programs that do not use `letcc` or `throw` (whether this actually makes a difference is for you to determine).

Submit the rules on paper or as a file `rules.txt`.

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

Task: Escaping Exceptions: Transition Rules (EXTRA CREDIT, 5 points)

Modify your transition rules above so that, if an exception is about to escape its scope—either by being raised, or by being returned as part of the body of the `exception` block—the machine enters the `escape` state:

$$\ldots \mapsto_{\mathsf{c}} \texttt{escape}$$

As before, your rules need to work correctly only in programs that do not use `letcc/throw`.

Submit the rules as a file `extra-1.txt` or on paper.

Task: Escaping Exceptions: Implementation (EXTRA CREDIT, 10 points)

Implement your rules that prevent exceptions from escaping, and submit the file as

`c-mach-no-escape.sml`

If you need a different definition of the `Exn` constructor, make the appropriate changes to `minml.sml` and submit it as

`minml-no-escape.sml`

Task: Escaping Exceptions: *Real* Implementation (EXTRA CREDIT, 10 points)

Would it be practical to keep exceptions from escaping in actual compiled code? Identify and discuss the issues that arise.

Submit your answer as a file `extra-2.txt` or on paper.

Test Cases

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. See below for submission instructions.

7 Hand-in Instructions

Turn in the files `minml.sml`, `parse.sml`, `typing.sml`, and `c-mach.sml` 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 files in the handin directory. Or, if you wish, you may turn in answers on paper, due in WeH 1313 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`