# CS520 Programming Assignment 2

**Posted: 26 Sept 2006**
**Due: 11:59pm, 17 Oct 2006**

**Overview** The purposes of this assignment are:

1. Implement a type-checker for a simply typed functional language named $\lambda_t$.
2. Implement an evaluator for $\lambda_t$.

**Syntax of $\lambda_t$**

| | | | |
|---|---|---|---|
| types | $ty$ | ::= | $\texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid ty_1 \rightarrow ty_2 \mid ty_1 * \ldots * ty_n$ |
| contants | $c$ | ::= | $() \mid \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid \ldots$ |
| operators | $op$ | ::= | $+ \mid - \mid * \mid / \mid \sim \mid \texttt{print}$ |
| terms | $t$ | ::= | $c \mid x \mid \texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid op(t_1, \ldots, t_n) \mid \texttt{lam}\ (x : ty) => t$ |
| | | | $\mid t_1(t_2) \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \mid \texttt{letrec}\ x : ty = t_1\ \texttt{in}\ t_2$ |
| | | | $\mid (t_0, \ldots, t_n) \mid t.i \mid \texttt{fix}(t) \mid (t : ty)$ |

**Operators** We assume the following operators in $\lambda_t$ with corresponding types:

| | | | |
|---|---|---|---|
| $+$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{int}$ | |
| $-$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{int}$ | |
| $*$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{int}$ | |
| $/$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{int}$ | |
| $\sim$ | : | $\texttt{int} \rightarrow \texttt{int}$ | (* negation *) |
| $>$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$ | |
| $>=$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$ | |
| $<$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$ | |
| $<=$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$ | |
| $=$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$ | |
| $<>$ | : | $\texttt{int} * \texttt{int} \rightarrow \texttt{bool}$ | |
| print | : | $\texttt{string} \rightarrow \texttt{unit}$ | |

**Abstract Syntax Definition of $\lambda_t$ in Ocaml**

```
type stp =
    TpBase of string   (* base type *)
  | TpFun of stp * stp (* function type *)
  | TpTup of stp list  (* tuple type *)

type ttm =
    TtmBool of bool    (* boolean constant *)
  | TtmInt of int      (* integer constant *)
  | TtmStr of string   (* string constant *)
  | TtmVar of string   (* variable *)
  | TtmIf of ttm * ttm * ttm   (* if-then-else term *)
  | TtmOp of string * ttm list (* built-in operator *)
  | TtmLam of string * stp * ttm (* lambda abstraction *)
  | TtmApp of ttm * ttm          (* application *)
  | TtmLet of string * ttm * ttm (* let-binding *)
  | TtmLetrec of string * stp * ttm * ttm (* letrec-binding *)
  | TtmTup of ttm list  (* tuple *)
  | TtmPro of ttm * int (* projection *)
  | TtmFix of ttm       (* fixed point *)
  | TtmAsc of ttm * stp (* ascription *)
```

**Static Semantics of $\lambda_t$**

$$\frac{c \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash c : \texttt{bool}} \ (\textbf{ty-bool})$$

$$\frac{c \in \{0, 1, 2, \ldots\}}{\Gamma \vdash c : \texttt{int}} \ (\textbf{ty-int})$$

$$\frac{}{\Gamma \vdash () : \texttt{unit}} \ (\textbf{ty-unit})$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \ (\textbf{ty-var})$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \texttt{lam}(x : T_1) => t : T_1 \to T_2} \ (\textbf{ty-lam})$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \ \ \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1(t_2) : T_2} \ (\textbf{ty-app})$$

$$\frac{\Sigma(op) = (T_1, \ldots, T_n) \to T \ \ \Gamma \vdash t_1 : T_1 \ \ \ldots \Gamma \vdash t_n : T_n}{\Gamma \vdash op(t_1, \ldots, t_n) : T} \ (\textbf{ty-op})$$

$$\frac{\Gamma \vdash t_0 : \texttt{bool} \ \ \Gamma \vdash t_1 : T \ \ \Gamma \vdash t_2 : T}{\Gamma \vdash \texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2 : T} \ (\textbf{ty-if})$$

$$\frac{\Gamma \vdash t_1 : T_1 \ \ \Gamma, x : T_1 \vdash t_2 : T}{\Gamma \vdash \texttt{let } x = t_1 \texttt{ in } t_2 : T} \ (\textbf{ty-let})$$

$$\frac{\Gamma, x : T_1 \vdash t_1 : T_1 \ \ \Gamma, x : T_1 \vdash t_2 : T}{\Gamma \vdash \texttt{letrec } x : T_1 = t_1 \texttt{ in } t_2 : T} \ (\textbf{ty-letrec})$$

$$\frac{\Gamma \vdash t_1 : T_1 \ \ \ldots \ \ \Gamma \vdash t_n : T_n}{\Gamma \vdash (t_1, \ldots, t_n) : T_1 * \ldots * T_n} \ (\textbf{ty-tup})$$

$$\frac{\Gamma \vdash t : T_1 * \ldots * T_n \ \ i = 1, \ldots, n}{\Gamma \vdash t.i : T_i} \ (\textbf{ty-proj})$$

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \texttt{fix}(t) : T} \ (\textbf{ty-fix})$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash (t : T) : T} \ (\textbf{ty-asc})$$

**Dynamic Semantics of $\lambda_t$**

$$\frac{t_0 \rightarrow t_0'}{\texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2 \rightarrow \texttt{if } t_0' \texttt{ then } t_1 \texttt{ else } t_2} \text{ (\textbf{eval-if})}$$

$$\frac{}{\texttt{if true then } t_1 \texttt{ else } t_2 \rightarrow t_1} \text{ (\textbf{eval-if-true})}$$

$$\frac{}{\texttt{if false then } t_1 \texttt{ else } t_2 \rightarrow t_2} \text{ (\textbf{eval-if-false})}$$

$$\frac{t_i \rightarrow t_i'}{op(v_1, \ldots, v_{i-1}, t_i, \ldots, t_n) \rightarrow op(v_1, \ldots, v_{i-1}, t_i', \ldots, t_n)} \text{ (\textbf{eval-op})}$$

$$\frac{op(v_1, \ldots, v_n) = v}{op(v_1, \ldots, v_n) \rightarrow v} \text{ (\textbf{eval-op-val})}$$

$$\frac{t_1 \rightarrow t_1'}{t_1(t_2) \rightarrow t_1'(t_2)} \text{ (\textbf{eval-app1})}$$

$$\frac{t_2 \rightarrow t_2'}{v_1(t_2) \rightarrow v_1(t_2')} \text{ (\textbf{eval-app2})}$$

$$\frac{}{(\texttt{lam}(x) \texttt{=>} t)(v) \rightarrow t[x \mapsto v]} \text{ (\textbf{eval-beta})}$$

$$\frac{t_i \rightarrow t_i'}{(v_1, \ldots, v_{i-1}, t_i, \ldots, t_n) \rightarrow (v_1, \ldots, v_{i-1}, t_i', \ldots, t_n)} \text{ (\textbf{eval-tup})}$$

$$\frac{}{(v_1, \ldots, v_n).i \rightarrow v_i} \text{ (\textbf{eval-proj})}$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{let } x = t_1 \texttt{ in } t_2 \rightarrow \texttt{let } x = t_1' \texttt{ in } t_2} \text{ (\textbf{eval-let})}$$

$$\frac{}{\texttt{let } x = v_1 \texttt{ in } t_2 \rightarrow t_2[x \mapsto v_1]} \text{ (\textbf{eval-let-val})}$$

$$\frac{t \rightarrow t'}{\texttt{fix}(t) \rightarrow \texttt{fix}(t')} \text{ (\textbf{eval-fix})}$$

$$\frac{}{\texttt{fix}(\texttt{lam}(x) \texttt{=>} t) \rightarrow t[x \mapsto \texttt{fix}(\texttt{lam}(x) \texttt{=>} t)]} \text{ (\textbf{eval-beta})}$$

$$\frac{}{\texttt{letrec } x : T = t_1 \texttt{ in } t_2 \rightarrow \texttt{let } x = \texttt{fix}(\texttt{lam}(x : T) \texttt{=>} t_1) \texttt{ in } t_2} \text{ (\textbf{def-letrec})}$$

**Problem 1 (100pts):** Based on the given static semantics, implement a function called `typecheck` in Ocaml which performs type checking for a $\lambda_t$ term. The `typecheck` function

should be assigned the following type in Ocaml:

`typecheck : ttm` $\rightarrow$ `stp option`

Note that for a well-typed term $t$ of type $T$, `typecheck`$(t)$ should return `Some`$(T)$; Otherwise, return `None`.

**Problem 2 (80pts):** Based on the given dynamic semantics, implement a function called `eval` in Ocaml which *evaluates closed well-typed* $\lambda_t$ *terms through the call-by-value strategy.* The `eval` function should have type

`eval : ttm` $\rightarrow$ `ttm`

in Ocaml.

**Implementation notes** A few files (in **prog2.tar.gz**) are provided to start the assignment. You need to provide the actual implementations of the above functions based on the given code. Once all the code are ready, type **make** under the directory. If no error reported, an executable file called **evaluator** will be produced. You can test your code by typing

```
./evaluator filename
```

where *filename* should be replaced by some actual file path.

**Grading** The grading of the assignment is based on whether the required functionalities are correctly implemented. Please make sure your code can be compiled and tested on **csa2** because all submissions will be tested on **csa2**. There are **20pts** for

1. if the code is well organized.
2. if errors are properly handled.
3. if the code has necessary comments.
4. etc.