1999

p3q4
AM

Compiler Construction

Compiling Techniques `cmptech1.tex`

A programming language has expressions e with the following syntax:

$$\begin{aligned} e ::= x \mid n \mid e + e' \mid e(e') \mid (e) \\ \mid \texttt{let}\ x = e\ \texttt{in}\ e' \\ \mid \texttt{letsta}\ f(x) = e\ \texttt{in}\ e' \\ \mid \texttt{letdyn}\ f(x) = e\ \texttt{in}\ e' \end{aligned}$$

where f and x range over identifiers and n ranges over numbers. The three *let* variants introduce simple variables (`let`) and (non-recursive) functions whose variables are statically (`letsta`) or dynamically (`letdyn`) bound.

Using e itself (or any related language whose relationship to e is explained) as abstract syntax define an evaluator *eval* which, when given an expression e and an environment ρ, yields the value of evaluating e in ρ. The evaluator can be written in a language of your choice or in mathematical pseudo-code.

Explain carefully in one sentence each:

- the forms of value which *eval* may return;
- the form(s) of value which constitute the environment;
- the use(s) of environment(s) in `letsta` and in a call to a function defined by `letsta`;
- the use(s) of environment(s) in `letdyn` and in a call to a function defined by `letdyn`.

[20 marks]

Hint: because both `letsta` and `letdyn` functions may be applied using the same function call syntax, you may find it helpful to use separate forms of value for the two forms of functions.

Model Answer

```
datatype Expr = Ide of string |
                Numb of int |
                Plus of Expr * Expr |
                Apply of Expr * Expr |
                Let of string * Expr * Expr |
                Letsta of string * string * Expr * Expr |
                Letdyn of string * string * Expr * Expr;

datatype Env =  Empty | Defn of string * Val * Env
and      Val =  IntVal of int |
                FnStaVal of string * Expr * Env |
                FnDynVal of string * Expr;

exception oddity of string;
```

```
fun lookup(n, Defn(x, v, r)) = if x=n then v else lookup(n, r)
  | lookup(n, Empty) = raise oddity("unbound name");

fun eval(Ide(x), r) = lookup(s, r)
  | eval(Numb(n), r) = IntVal(n)
  | eval(Plus(e, e'), r) =
      let val v = eval(e,r);
          val v' = eval(e',r)
      in case (v,v') of (IntVal(i), IntVal(i')) => IntVal(i+i')
                      | (v, v') => raise oddity("plus of non-number")
      end
  | eval(Apply(e, e'), r) =
      (case eval(e, r) of IntVal(i) => raise oddity("apply of non-function")
                        | FnStaVal(bv, body, r_fromdef) =>
                             let val arg = eval(e', r)
                             in eval(body, Defn(bv, arg, r_fromdef))
                             end
                        | FnDynVal(bv, body) =>
                             let val arg = eval(e', r)
                             in eval(body, Defn(bv, arg, r))
                             end
                        )
  | eval(Let(x, e, e'), r) =
      eval(e', Defn(x, eval(e, r), r))
  | eval(Letsta(f, x, e, e'), r) =
      eval(e', Defn(f, FnStaVal(x,e,r), r))
  | eval(Letsta(f, x, e, e'), r) =
      eval(e', Defn(f, FnDynVal(x,e), r));
```

Individual points:
1. **eval** may return values which are (1) integers, (2) closures representing statically bound fns or (3) closures without environments representing statically bound fns.
2. Environments are either empty or consist of a list of (name, value[from point 1]) pairs.
3. letsta fns preserve the defining env in FnStaVal and use it (r_fromdef) in the call for evaluating free variables of the function. 4. letdyn fns ignore the defining env when forming FnVal and use the env existing at time of call for evaluating free variables of the function.