

Exercise sheet 5 for 13 March 2002

2002-03-06

Do exercises 5.1, 5.2, 5.3, 5.4, 5.5. Hand in the solutions.

Exercise 5.1 Define the following polymorphic SML functions on lists using the `foldr` function for lists:

- `filter : ('a -> bool) -> ('a list -> 'a list)`
where `filter p xs` applies `p` to all elements `x` of `xs` and returns a list of those for which `p x` is true.
- `mapPartial : ('a -> 'b option) -> ('a list -> 'b list)`
where `mapPartial f xs` applies `f` to all elements `x` of `xs` and returns a list the values `y` for which `f x` has form `SOME y`.
Thus `mapPartial (fn i => if i>7 then SOME(i-7) else NONE) [4, 12, 3, 17, 10]` should give `[5, 10, 3]`.

Exercise 5.2 Define the polymorphic SML function `tmap : ('a -> 'b) -> ('a tree -> 'b tree)` so that `tmap f t` creates a new tree of the same shape as `t`, but in which the value of a node is `f v` if the value of the corresponding node in the old tree is `v`.

Exercise 5.3 Define the following polymorphic SML functions on trees using the `tfold` function from the lecture:

- `preorder1 : 'a tree -> 'a list`
- `inorder1 : 'a tree -> 'a list`
- `postorder1 : 'a tree -> 'a list`
- the above function `tmap`
- the function `depth : 'a tree -> int` from exercise sheet 1.

Recall that the `preorder`, `inorder`, and `postorder` traversals were defined in exercise sheet 2.

Exercise 5.4 Add anonymous functions, similar to SML's `fn x => ...`, to the higher-order functional language:

```
datatype expr =
  ...
  | Fn of string * expr
  | ...
```

Anonymous functions give rise to non-recursive closures of the form

```
datatype value =
  ...
  | Clo of string * expr (* (x, body, bodyenv) *)
```

For instance, the expression `fn x => 2*x` might evaluate to this closure:

`Clo("x", Prim("**", [CstI 2, Var "x"]), [])` in the empty environment `Env.empty`.

Similarly, the expression `let y = 22 in fn z => z+y end` might evaluate to this closure:

`Clo("z", Prim("+", [Var "z", Var "y"]), [y ↦ 22])`.

Extend the evaluator in file `fun/hofun.sml` to interpret such anonymous functions.

Exercise 5.5 Extend the lexer and parser specification to permit anonymous functions. The concrete syntax may be as in SML: `fn x => expr`, where `x` is a variable.

Exercise 5.6 Write an SML function `check : expr -> bool` that checks whether all variables and function names are defined when they are used. This checker should accept the higher-order language. That is, in the abstract syntax `Call(e1, e2)` for a function call, the expression `e1` can be an arbitrary expression and need not be a variable name.

Exercise 5.7 Add mutually recursive function declarations in the higher-order functional language:

```
datatype expr =
  ...
  | Letfun of fundef list * expr
  | ...

wheretype fundef = string * string * expr
```

Exercise 5.8 Add lists (`CstN` is the empty list `CstN`, `ConC(e1, e2)` is `e1::e2`), and case expressions to the language, where `Case(e0, e1, ("h","t", e2))` is `case e0 of [] => e1 | h::t => e2`:

```
datatype expr =
  ...
  | CstN
  | ConC of expr * expr
  | Case of expr * expr * (string * string * expr)
  | ...
```

Exercise 5.9 Implement some higher-order functions on lists as examples. You may get inspiration from the structure `List` in `Moscow ML` or `Standard ML Basis Library`.

Exercise 5.10 Design a concrete syntax for the higher-order functional language extended with lists, extend the lexer and parser specifications, and write some example programs in concrete syntax.

Exercise 5.11 Extend the (monomorphic or polymorphic) type checker to deal with lists. Use the following extra kinds of types:

```
datatype typ =
  ...
  | TypL of typ (* list, element type is typ *)
  | ...
```

Exercise 5.12 Study a lazy functional language such as Haskell (www.haskell.org). The Haskell compiler that is easiest to install and use is probably Hugs (<http://www.haskell.org/hugs/>).