

## Exercise sheet 4 for 5 March 2003

2003-02-26

Do exercises 4.1, 4.2, 4.3, 4.4, 4.5. Hand in solutions to all except exercise 4.1.

**Exercise 4.1** In directory `fun`, compile the `Env.sig` and `Env.sml` files using `mosmlc -c`, then generate and compile the lexer and parser for the small functional language as described in `grammar.txt`. Parse and run some example programs from file `fun/parse.sml`.

**Exercise 4.2** Write more example programs in the functional language. For instance, write programs that do the following:

- Compute the sum of the numbers from 1000 down to 1. Do this by defining a function `sum n` that computes  $n + (n - 1) + \dots + 2 + 1$  although there are smarter ways to do this.
- Compute the number  $3^8$ , that is, 3 raised to the power 8. Again, use a recursive function.
- Compute  $3^0 + 3^1 + \dots + 3^{10} + 3^{11}$ , using two recursive functions.
- Compute  $1^8 + 2^8 + \dots + 10^8$ , again using two recursive functions.

**Exercise 4.3** For simplicity, the current implementation of the functional language requires all functions to take exactly one argument. This seriously limits the programs that can be written in the language (at least it limits what that can be written without excessive cleverness and complications).

Modify the language to permit functions to take one or more arguments. Start by modifying the abstract syntax in `fun/Absyn.sml` to permit a list of parameter names in `Letfun` and a list of argument expressions in `Call`. After doing so you must compile it with `mosmlc -c Absyn.sml`, shut down `mosml` and then restart it to load the new compiled version of `Absyn`.

Then modify the `eval` interpreter in file `fun/fun.sml` to work for the new abstract syntax. You must modify the closure representation to accommodate a list of parameters. Also, modify the `Letfun` and `Call` clauses of the interpreter. You will probably find the `bindZip` function from the `Env` structure useful.

**Exercise 4.4** In continuation of Exercise 4.3, modify the parser specification to accept a language where functions may take any (non-zero) number of arguments. The resulting parser should permit function declarations such as these:

```
let pow x n = if n=0 then 1 else x * pow x (n-1) in pow 3 8 end

let max2 a b = if a<b then b else a
in let max3 a b c = max2 a (max2 b c)
   in max3 25 6 62 end
end
```

You may want to define non-empty parameter lists and argument lists in analogy with the `Names1` nonterminal from `usql/Sqlpar.grm`, except that the parameters are not separated by commas.

**Exercise 4.5** Extend the (untyped) functional language with sequential logical ‘and’ (like SML’s `andalso` and Java’s `&&`) and with sequential logical ‘or’ (like SML’s `orelse` and Java’s `||`). Notice that `e1 && e2` can be encoded as `if e1 then e2 else false` and that `e1 || e2` can be encoded as `if e1 then true else e2`. Hence you need only change the lexer and parser specifications, and make the new rules in the parser specification generate the appropriate abstract syntax. You need not change `fun/Absyn.sml` or `fun/fun.sml`.

**Exercise 4.6** Extend the abstract syntax, the concrete syntax, and the interpreter to handle tuple constructors ( . . . ) and component selectors #i:

```
datatype expr =
  ...
  | Tup of expr list
  | Sel of int * expr
  | ...
```

If we use the concrete syntax #2(e) for Sel(2, e) and (e1, e2) for Tup[e1, e2] then you should be able to write programs such as these:

```
let t = (1+2, false, 5>8)
in if #3(t) then #1(t) else 14 end
```

and

```
let max xy = if #1(xy) > #2(xy) then #1(xy) else #2(xy)
in max (3, 88) end
```

Note that this permits functions to take multiple arguments and return multiple results.

To extend the interpreter correspondingly, you need to introduce a new kind of value, namely a tuple value `TupV(vs)`, and to allow `eval` to return a result of type `value` (not just an integer):

```
datatype value =
  Int of int
  | TupV of value list
  | RClo of string * string list * expr * vfenv
withtype vfenv = (string, value) env

fun eval (e : expr) (env : vfenv) : value = ...
```

Note that this requires some changes elsewhere in the `eval` interpreter. For instance, the primitive operations currently work because `eval` always returns an `int`, but with the suggested change, they will have to check that `eval` returns an `Int i` (e.g. by pattern matching).

**Exercise 4.7** Add tuples and selectors to the typed functional language (`fun/tychk.sml`), and implement type checking for tuples and selectors also. You will need to introduce a new kind of type to represent the type of a tuple of values:

```
datatype typ =
  ...
  | TypT of typ list (* tuple; list of element types *)
```

so that the tuple `t` in Exercise 4.6 would have type `TupT[TypI, TypB, TypB]`, and function `max` would have type `TypF(TypT[TypI, TypI], TypI)`.

**Exercise 4.8** Add lists (`CstN` is the empty list `[]`, `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 4.9** Add type checking for lists. All elements of a list must have the same type. You'll need a new kind of type `TypL of typ` to represent the type of lists with elements of a given type.