Appendix E

Supporting code for the ML interpreter for μScheme

E.1 Tokens of the μScheme language

Our general parsing mechanism from Appendix D requires us to define a `token` type and two functions `tokenString` and `isLiteral`.

671a ⟨*lexical analysis* 671a⟩≡ (224) 671b▷

```
datatype token = NAME    of string
               | INT     of int
               | SHARP   of bool
               | BRACKET of char (* ( or ) *)
               | QUOTE
```

I define `isLiteral` by comparing the given string `s` with the string form of token `t`.

671b ⟨*lexical analysis* 671a⟩+≡ (224) ◁671a 672a▷

```
fun tokenString (NAME x)    = x
  | tokenString (INT  n)    = Int.toString n
  | tokenString (SHARP b)   = if b then "#t" else "#f"
  | tokenString (BRACKET c) = str c
  | tokenString (QUOTE)     = "'"

fun isLiteral s t = tokenString t = s
⟨support for streams, lexical analysis, and parsing 644⟩
```

Before a μScheme token, whitespace is ignored. The `schemeToken` function enumerates the alternatives: two brackets, a quote mark, an integer literal, an atom, or end of line.

672a ⟨*lexical analysis* 671a⟩+≡ (224) ◁671b

`schemeToken : token lexer`
`atom : string -> token`

```
local
  ⟨functions used in the lexer for μScheme 672b⟩
in
  val schemeToken =
    whitespace *> (   BRACKET <$> oneEq #"("
                  <|> BRACKET <$> oneEq #")"
                  <|> QUOTE   <$  oneEq #"'"
                  <|> INT     <$> intToken isDelim
                  <|> (atom o implode) <$> many1 (sat (not o isDelim) one)
                  <|> noneIfLineEnds
                  )
end
```

The `atom` function identifies the special literals `#t` and `#f`; all other atoms are names.

672b ⟨*functions used in the lexer for μScheme* 672b⟩≡ (672a) 672c▷

```
fun atom "#t" = SHARP true
  | atom "#f" = SHARP false
  | atom x    = NAME x
```

If the lexer doesn't recognize a bracket, quote mark, integer, or other atom, we're expecting the line to end. The end of the line may present itself as the end of the input stream or as a stream of characters beginning with a semicolon, which marks a comment. If we encounter any other character, something has gone wrong. (The polymorphic type of `noneIfLineEnds` provides a subtle but powerful hint that no token can be produced; the only possible outcomes are that nothing is produced, or the lexer detects an error.)

⟨*functions used in the lexer for μScheme* 672b⟩+≡ (672a) ◁672b

`noneIfLineEnds : 'a lexer`

```
fun noneIfLineEnds chars =
  case streamGet chars
    of NONE => NONE (* end of line *)
     | SOME (#";", cs) => NONE (* comment *)
     | SOME (c, cs) =>
         let val msg = "invalid initial character in `" ^
                       implode (c::listOfStream cs) ^ "'"
         in  SOME (ERROR msg, EOS)
         end
```


E.2 Parsing

A parser consumes a stream of tokens and produces an abstract-syntax tree. The easiest way to write a parser is to begin with code for parsing the smallest things and finish with the code for parsing the biggest things. I parse tokens, literal S-expressions, μScheme expressions, and finally μScheme definitions.

Usually a parser knows what kind of token it is looking for. To make such a parser easier to write, I create special parsing combinators for each kind of token. Each one succeeds when given a token of the kind it expects; when given any other token, it fails.

673a ⟨*parsing* 673a⟩≡ (224) 673b▷

```
val name    = (fn (NAME  n) => SOME n  | _ => NONE) <$>? token
val booltok = (fn (SHARP b) => SOME b  | _ => NONE) <$>? token
val int     = (fn (INT   n) => SOME n  | _ => NONE) <$>? token
val quote   = (fn (QUOTE)   => SOME () | _ => NONE) <$>? token
```

I'm now ready to parse a quoted S-expression, which is a symbol, a number, a Boolean, a list of S-expressions, or a quoted S-expression.

673b ⟨*parsing* 673a⟩+≡ (224) ◁673a 673c▷

`sexp : value parser`

```
fun sexp tokens = (
     SYM           <$> (notDot <$>! name)
  <|> NUM          <$> int
  <|> BOOL         <$> booltok
  <|> embedList    <$> "(" >-- many sexp --< ")"
  <|> (fn v => embedList [SYM "quote", v])
                   <$> (quote *> sexp)
) tokens
and notDot "." = ERROR "this interpreter cannot handle . in quoted S-expressions"
  | notDot s   = OK s
```

Full Scheme allows programmers to notate arbitrary cons cells using a dot in a quoted S-expression. μScheme doesn't support this notation.

The next step up is syntactic elements used in expressions. Function `formals` parses a list of formal parameters. Function `lambda` forms a `LAMBDA` expression, *provided* there are no duplicate names among the formal parameters:

673c ⟨*parsing* 673a⟩+≡ (224) ◁673b 673d▷

```
formals : name list parser
lambda  : name list located -> exp -> exp error
```

```
val formals =
  "(" >-- many name --< ")"
fun lambda xs exp =
  nodups ("formal parameter", "lambda") xs >>=+ (fn xs => LAMBDA (xs, exp))
```

Function `letx` forms a `LETX` expression, provided there are no duplicates among the bound names—except when the `LETX` expression is `LETSTAR`, because duplicate names in `LETSTAR` are permissible.

673d ⟨*parsing* 673a⟩+≡ (224) ◁673c 674a▷

```
letx : let_kind -> (name * exp) list located -> exp -> exp error
```

```
local
  fun letDups LETSTAR (loc, bindings) = OK bindings
    | letDups kind    (loc, bindings) =
        let val names    = map (fn (n, _) => n) bindings
            val kindName = case kind of LET => "let" | LETREC => "letrec" | _ => "??
        in  nodups ("bound name", kindName) (loc, names) >>=+ (fn _ => bindings)
        end
in
  fun letx kind bs exp = letDups kind bs >>=+ (fn bs => LETX (kind, bs, exp))
end
```

Parsing function `exp` handles all the concrete syntax for μScheme expressions, which is shown in Section 3.11.1 on page 113. Most constructs of μScheme are notated using expressions bracketed in parentheses, for which purpose I use function `bracket` from page 665 in Appendix D. The word `bracket` takes up a bit of horizontal space, and I'm squeezing the code to try to fit each syntactic production on one line. So instead of writing `bracket` out in full, I define an abbreviation `br`.

674a ⟨*parsing* 673a⟩+≡ (224) ◁673d 674b▷

```
exp      : exp parser
bindings : (name * exp) list parser
```

```
val br = bracket
fun exp tokens = (
       VAR                 <$> name
  <|> (LITERAL o NUM)  <$> int
  <|> (LITERAL o BOOL) <$> booltok
  <|> LITERAL          <$> (quote *> sexp)
  <|> br "if"     "(if e1 e2 e3)"             (curry3 IFX     <$> exp  <*> exp <*> exp)
  <|> br "while"  "(while e1 e2)"             (curry  WHILEX  <$> exp  <*> exp)
  <|> br "set"    "(set x e)"                 (curry  SET     <$> name <*> exp)
  <|> br "begin"  ""                          (       BEGIN   <$> many exp)
  <|> br "lambda" "(lambda (names) body)"     (       lambda  <$> @@ formals  <*>! exp)
  <|> br "let"    "(let (bindings) body)"     (letx   LET     <$> @@ bindings <*>! exp)
  <|> br "letrec" "(letrec (bindings) body)"  (letx   LETREC  <$> @@ bindings <*>! exp)
  <|> br "let*"   "(let* (bindings) body)"    (letx   LETSTAR <$> @@ bindings <*>! exp)
  <|> "(" >-- literal ")" <!> "empty application"
  <|> curry APPLY <$> "(" >-- exp <*> many exp --< ")"
 ) tokens
and bindings ts = ("(" >-- (many binding --< ")" <?> "(x e)...")) ts
and binding  ts = ("(" >-- (pair <$> name <*> exp --< ")" <?> "(x e) in bindings")) ts
```

An expression can contain bindings, and bindings contain expressions, so functions `exp` and `bindings` must be mutually recursive.

Function `dfn` is a bit like `lambda`: it detects duplicate formal parameters. The name "dfn" allows me more horizontal space than I would have if I used "`define`."

674b ⟨*parsing* 673a⟩+≡ (224) ◁674a 674c▷

```
dfn : name -> name list located -> exp -> def error
```

```
fun dfn f formals body =
  nodups ("formal parameter", "definition of function " ^ f) formals >>=+
   (fn xs => DEFINE (f, (xs, body)))
```

Function `def` parses a definition.

674c ⟨*parsing* 673a⟩+≡ (224) ◁674b 674d▷

```
def : def parser
```

```
val def =
      bracket "define" "(define f (args) body)" (dfn <$> name <*> @@ formals <*>! exp)
  <|> bracket "val"    "(val x e)"              (curry VAL <$> name <*> exp)
  <|> bracket "use"    "(use filename)"         (USE       <$> name)
  <|> literal ")" <!> "unexpected right parenthesis"
  <|> EXP <$> exp
  <?> "definition"
```

Pair `schemeSyntax` contains the lexer and the parser.

674d ⟨*parsing* 673a⟩+≡ (224) ◁674c

```
schemeSyntax : token lexer * def parser
```

```
val schemeSyntax = (schemeToken, def)
```

E.3 Further reading

Koenig (1994) describes an experience with ML type inference which leads to a conclusion that resembles my conclusion about the type of `noneIfLineEnds` on page 672c.