Appendix I

Supporting code for μSmalltalk

I.1 Lexing and parsing

I.1.1 Lexical analysis

There are three reasons we can't reuse μScheme's lexer for μSmalltalk: μSmalltalk introduces literals with # instead of ', μSmalltalk treats square brackets as syntactic sugar for parameterless blocks, and μSmalltalk keeps track of source-code locations. Aside from these details, the lexers are the same.

A source-code location includes a name for the source, plus line number.

699a ⟨*lexical analysis* 699a⟩≡ (475a) 699b▷
```
type srcloc = string * int
val nullsrc = ("internally generated SEND node", 1)
fun srclocString (source, line) = source ^ ", line " ^ Int.toString line
```

The representation of a token is almost the same as in μScheme. The differences are that there are two kinds of brackets, and that a # character does not introduce a Boolean.

699b ⟨*lexical analysis* 699a⟩+≡ (475a) ◁699a 699c▷
```
datatype token = INT     of int
               | NAME    of name
               | BRACKET of char          (* ( or ) or [ or ] *)
               | SHARP   of string option (* symbol or array *)
```

To produce error messages, we must be able to convert a token back to a string.

699c ⟨*lexical analysis* 699a⟩+≡ (475a) ◁699b 699d▷
```
fun tokenString (INT     n)      = Int.toString n
  | tokenString (NAME    x)      = x
  | tokenString (BRACKET c)      = str c
  | tokenString (SHARP NONE)     = "#"
  | tokenString (SHARP (SOME s)) = "#" ^ s
```

699d ⟨*lexical analysis* 699a⟩+≡ (475a) ◁699c 700a▷
```
fun isLiteral s token =
  case (s, token) of ("#", SHARP NONE) => true
                   | (s,   NAME s' )   => s = s'
                   | (s,   BRACKET c)  => s = str c
                   | _ => false
```
⟨*support for streams, lexical analysis, and parsing* 644⟩

700a ⟨*lexical analysis* 699a⟩+≡ (475a) ◁699d

smalltalkToken : token lexer

```
local
  val isDelim = fn c => isDelim c orelse c = #"[" orelse c = #"]"
  val nondelims = many1 (sat (not o isDelim) one)

  fun validate NONE = NONE (* end of line *)
    | validate (SOME (#";", cs)) = NONE (* comment *)
    | validate (SOME (c, cs)) =
        let val msg = "invalid initial character in `" ^
                      implode (c::listOfStream cs) ^ "'"
        in  SOME (ERROR msg, EOS) : (token error * char stream) option
        end
in
  val smalltalkToken =
    whitespace *> (
          BRACKET                  <$> sat (Char.contains "()[]") one
        <|> (SHARP o SOME o implode) <$> (oneEq #"#" *> nondelims)
        <|> SHARP NONE               <$  oneEq #"#"
        <|> INT                      <$> intToken isDelim
        <|> (NAME o implode)         <$> nondelims
        <|> (validate o streamGet)
        )
end
```

The `isDelim` on the *right* of `val isDelim` is from ⟨*lexing support* (generated automatically)⟩. The `val isDelim` introduces a *new* `isDelim` that recognizes square brackets as delimiters.

<$> 653c
<|> 654b
BRACKET 699b
EOS 647a
ERROR 651a
type error 651a
INT 699b
intToken 660c
isDelim 659b
listOfStream 647d
many1 658a
NAME 699b
nodups 666a
one 655b
oneEq 656c
sat 656b
SHARP 699b
type stream 647a
streamGet 647b
whitespace 659c

I.1.2 Parsing

⟨*parsing* 700b⟩≡ (475a) 701▷

```
val blockDups         = nodups ("formal parameter",  "block")
fun methodDups kind f = nodups ("formal parameter",   kind ^ " " ^ f)
fun localDups  kind f = nodups ("local variable",     kind ^ " " ^ f)
fun ivarDups c        = nodups ("instance variable", "class " ^ c)
```

Smalltalk has simple rules for computing the arity of a message based on the message's name: if the name is symbolic, the message is binary (one receiver, one argument); if the name is alphanumeric, the number of arguments is the number of colons. Unfortunately, in μSmalltalk a name can mix alphanumerics and symbols. To decide the issue, we use the *first* character of a message's name.

701 ⟨*parsing* 700b⟩+≡ (475a) ◁700b 702▷

```
fun arity "if"    = 2
  | arity "while" = 1
  | arity name =
      let val cs = explode name
      in  if Char.isAlpha (hd cs) then
            length (List.filter (fn c => c = #":") cs)
          else
            1
      end

fun arityOk "value" _ = true
  | arityOk name args = arity name = length args

fun arityErrorAt loc what msgname args =
  let fun argn n = if n = 1 then "1 argument" else Int.toString n ^ " arguments"
  in  errorAt ("in " ^ what ^ ", message " ^ msgname ^ " expects " ^
                         argn (arity msgname) ^ ", but gets " ^
                         argn (length args)) loc
  end
```

errorAt 661b

Here's the parser.

702 ⟨*parsing* 700b⟩+≡ (475a) ◁701 703▷

```
val name = (fn (NAME s)         => SOME s | _ => NONE) <$>? token
val int  = (fn (INT n)          => SOME n | _ => NONE) <$>? token
val sym  = (fn (SHARP (SOME s)) => SOME s | _ => NONE) <$>? token

fun isImmutable x =
  List.exists (fn x' => x' = x) ["true", "false", "nil", "self", "super"]
val immutable = sat isImmutable name

val mutable =
  let fun can'tMutate (loc, x) =
        ERROR (srclocString loc ^ ": you cannot set or val-bind pseudovariable " ^ x)
  in  can'tMutate <$>! @@ immutable <|> OK <$>! name
  end

val formals  = "(" >-- many name --< ")"
val bformals = blockDups <$>! @@ formals

fun exp tokens = (
  (* must be allowed to fail since it is used in 'many exp' *)
      (LITERAL o NUM)    <$> int
  <|> (LITERAL o SYM)    <$> sym
  <|> SUPER              <$  literal "super"
  <|> VAR                <$> name
  <|> literal "#" *> (   (LITERAL o SYM)                <$> name
                     <|> (LITERAL o SYM o Int.toString) <$> int
                     <|> VALUE                          <$> sharp
                            (* last better not happen in initial basis *)
                     )
  <|> bracket "set"   "(set x e)"             (curry SET   <$> mutable <*> exp)
  <|> bracket "begin" "(begin e ...)"         (      BEGIN <$> many exp)
  <|> bracket "block" "(block (x ...) e ...)" (curry BLOCK <$> bformals <*> many exp)
  <|> bracket "locals" "expression"
                (errorAt "found '(locals ...)' where an expression was expected"
                 <$>! srcloc)
  <|> curry BLOCK [] <$> "[" >-- many exp  --< "]"
  <|> messageSend    <$> "(" >-- @@ name <*> exp <*>! many exp --< ")"
  <|> (fn (loc, n) => errorAt ("sent message " ^ n ^ " to no object") loc) <$>!
      "(" >-- @@ name --< ")"
  <|> "(" >-- literal ")" <!> "empty message send ()"
  )
  tokens
and messageSend (loc, msgname) receiver args =
      if arityOk msgname args then
          OK (SEND (loc, msgname, receiver, args))
      else
          arityErrorAt loc "message send" msgname args
```

name : string parser
int : int parser

If any μSmalltalk code tries to change any of the predefined "pseudovariables," the `settable` parser causes an error.

The remaining parser functions are mostly straightforward. The `sharp` function may call `mkSymbol`, `mkInteger`, or `mkArray`, which must not be called until after the initial basis is read in.

703 ⟨*parsing* 700b⟩+≡ (475a) ◁702 704▷

```
and sharp tokens = (
        mkSymbol  <$> name
    <|> mkInteger <$> int
    <|> mkArray   <$> ("(" >-- many sharp --< ")")
    <|> literal "#" <!> "# within # is not legal"
    <|> literal "[" <!> "[ within # is not legal"
    <|> literal "]" <!> "] within # is not legal"
    ) tokens
```

`--<`	664c
`<!>`	664a
`<$>`	653c
`<|>`	654b
`>--`	664c
`int`	702
`literal`	664b
`many`	657d
`mkArray`	476c
`mkInteger`	476c
`mkSymbol`	476c
`name`	702

The parser for definitions recognizes `method` and `class-method`, because if a class definition has an extra right parenthesis, a `method` or `class-method` keyword might show up at top level.

704 ⟨*parsing* 700b⟩+≡ (475a) ◁703 705a▷

```
type 'a located = srcloc * 'a

fun def tokens = (
     bracket "define" "(define f (args) body)"
                 (curry3 DEFINE <$> name <*> formals <*> exp)
  <|> bracket "class" "(class name super (instance vars) methods)"
                 (classDef      <$> name <*> name <*> @@ formals <*>! many method)
  <|> bracket "val"   "(val x e)"      (curry VAL <$> mutable <*> exp)
  <|> bracket "use"   "(use filename)" (      USE <$> name)
  <|> bracket "method" ""              (badDecl "method")
  <|> bracket "class-method" ""        (badDecl "class-method")
  <|> literal ")" <!> "unexpected right parenthesis"
  <|> EXP <$> exp
  <?> "definition"
  ) tokens

and classDef name super ivars methods =
      (ivarDups name ivars) >>=+ (fn ivars =>
      CLASSD { name = name, super = super, ivars = ivars, methods = methods })

and method tokens =
  let datatype ('a, 'b) imp = PRIM of 'a | USER of 'b
      val user : string list located * string list located * exp list ->
                 (string, string list located * string list located * exp list) imp = USER
      fun imp kind =  PRIM <$> "primitive" >-- name
                 <|> curry3 USER <$> @@ formals <*> @@ locals <*> many exp
      and locals tokens =
            (bracket "locals" "(locals ivars)" (many name) <|> pure []) tokens

      fun method kind name impl =
        check (kname kind, name, impl) >>= (fn impl => OK (kind, name, impl))
      and kname IMETHOD = "method"
        | kname CMETHOD = "class-method"
      and check (_, _, PRIM p) = OK (PRIM_IMPL p)  (* no checking possible *)
        | check (kind, name, USER (formals as (loc, _), locals, body)) =
            methodDups kind name formals >>= (fn formals =>
            localDups  kind name locals  >>= (fn locals =>
                if arityOk name formals then
                  OK (USER_IMPL (formals, locals, BEGIN body))
                else
                  arityErrorAt loc (kind ^ " definition") name formals))

      val name' = (fn n => ((*app print ["Parsing method ", n, "\n"];*) n)) <$> name
  in   bracket "method"       "(method f (args) body)"
                                    (method IMETHOD <$> name' <*>! imp "method")
   <|> bracket "class-method" "(class-method f (args) body)"
                                    (method CMETHOD <$> name' <*>! imp "class method")
  end tokens
```


<!> 664a
<$> 653c
<$>! 658c
<*> 653b
<*>! 658c
<?> 663c
<|> 654b
>-- 664c
>>= 651b
>>=+ 652a
arityErrorAt 701
arityOk 701
BEGIN 467a
bracket 665
CLASSD 467b
CMETHOD 467b
curry 654a
curry3 654a
DEFINE 467b
errorAt 661b
EXP 467b
type exp 467a
exp 702
formals 702
IMETHOD 467b
ivarDups 700b
literal 664b
localDups 700b
many 657d
methodDups 700b
mutable 702
name 702
OK 651a
PRIM_IMPL 467b
pure 653a
srcloc 663a
USE 467b
USER_IMPL 467b
VAL 467b

```
and badDecl what =
  errorAt ("unexpected `(" ^ what ^ "...'; " ^
           "did a class definition end prematurely?") <$>! srcloc
```

705a ⟨*parsing* 700b⟩+≡ (475a) ◁704

```
val smalltalkSyntax = (smalltalkToken, def)
```

I.2 Support for tracing

Tracing support is divided into three parts: support for printing indented messages, which is conditioned on the value of the variable `&trace`; support for maintaining a stack of source-code locations, which is used to provide information when an error occurs; and exposed tracing functions, which are used in the main part of the interpreter. To keep the details hidden from the rest of the interpreter, the first two parts are made `local`.

705b ⟨*tracing* 705b⟩≡ (486a) 707a▷

```
local
  ⟨private state and functions for printing indented traces 705c⟩
  ⟨private state and functions for mainting a stack of source-code locations 706b⟩
in
  ⟨exposed tracing functions 706c⟩
end
```

The `traceMe` function is used internally to decide whether to trace; it not only returns a Boolean but also decrements `&trace` if needed.

705c ⟨*private state and functions for printing indented traces* 705c⟩≡ (705b) 705d▷

```
fun traceMe xi =
  let val count = find("&trace", xi)
  in  case !count
        of (c, NUM n) =>
            if n = 0 then false
            else ( count := (c, NUM (n - 1))
                 ; if n = 1 then (print "<trace ends>\n"; false) else true
                 )
         | _ => false
  end handle NotFound _ => false
```

`traceMe : value ref env -> bool`

The local variable `tindent` maintains the current trace state; `indent` uses it to print an indentation string.

705d ⟨*private state and functions for printing indented traces* 705c⟩+≡ (705b) ◁705c 706a▷

```
val tindent = ref 0
fun indent 0 = ()
  | indent n = (print "  "; indent (n-1))
```

def	704
find	214
NotFound	214
NUM	468a
smalltalkToken	700a

Any actual printing is done by `tracePrint`, conditional on `traceMe` returning `true`. The argument `direction` of type `indentation` controls the adjustment of `indent`. For consistency, we outdent from the previous level *before* printing a message; we indent from the current level *after* printing a message.

706a ⟨*private state and functions for printing indented traces* 705c⟩+≡ (705b) ◁705d

```
datatype indentation = INDENT_AFTER | OUTDENT_BEFORE

fun tracePrint direction xi f =
    if traceMe xi then
      let val msg = f () (* could change tindent *)
      in  ( if direction = OUTDENT_BEFORE then tindent := !tindent - 1 else ()
          ; indent (!tindent)
          ; app print msg
          ; print "\n"
          ; if direction = INDENT_AFTER   then tindent := !tindent + 1 else ()
          )
      end
    else
        ()
```

Printing of trace messages is conditional, but we always maintain a stack of source-code locations. The stack is displayed when an error occurs.

706b ⟨*private state and functions for mainting a stack of source-code locations* 706b⟩≡ (705b)

```
val locationStack = ref [] : (string * srcloc) list ref
fun push srcloc = locationStack := srcloc :: !locationStack
fun pop () = case !locationStack
               of []     => raise InternalError "tracing stack underflows"
                | h :: t => locationStack := t
```

Here are the tracing-related functions that are exposed to the rest of the interpreter. The interpreter uses `traceIndent` to trace sends, `outdentTrace` to trace answers, and `resetTrace` to reset indentation.

706c ⟨*exposed tracing functions* 706c⟩≡ (705b)

```
resetTrace     : unit -> unit
traceIndent    : string * srcloc -> value ref env -> (unit -> string list) -> unit
outdentTrace   :                    value ref env -> (unit -> string list) -> unit
showStackTrace : unit -> unit
```

```
fun resetTrace ()       = (locationStack := []; tindent := 0)
fun traceIndent what xi = (push what; tracePrint INDENT_AFTER   xi)
fun outdentTrace     xi = (pop ();    tracePrint OUTDENT_BEFORE xi)
fun showStackTrace () =
  let fun show (msg, (file, n)) =
        app print ["  Sent '", msg, "' in ", file, ", line ", Int.toString n, "\n"]
  in  case !locationStack
        of [] => ()
         | l  => ( print "Method-stack traceback:\n"; app show (!locationStack) )
  end
```

indent 705d
InternalError 469
tindent 705d
traceMe 705c

To avoid confusion, tracing code typically avoids `print` methods; instead, it uses `valueString` to give information about a value.

707a ⟨*tracing* 705b⟩+≡ (486a) ◁705b

```
fun valueString (c, NUM n) =
      String.map (fn #"~" => #"-" | c => c) (Int.toString n) ^
      valueString(c, USER [])
  | valueString (_, SYM v) = v
  | valueString (c, _) = "<" ^ className c ^ ">"
```

I.3 Miscellaneous

We have a different version of `use` than in μScheme; `echo` is a Boolean, not a function.

707b ⟨*implementation of* `use`, *with Boolean* `echo` 707b⟩≡ (475a)

```
fun use readEvalPrint filename rho =
      let val fd = TextIO.openIn filename
          val defs = reader smalltalkSyntax noPrompts (filename, streamOfLines fd)
          fun errln s = TextIO.output (TextIO.stdErr, s ^ "\n")
      in  readEvalPrint (defs, true, errln) rho
          before TextIO.closeIn fd
      end
```