Main Driver for Compiler using `mosmllex` and `mosmlyac`

Franklin Chen

July 28, 1996

This is a driver that simply hooks up `Lexer` and `Parser`.

1a ⟨*** 1a⟩≡
```
local open ⟨Modules to open 1b⟩ in
  ⟨Auxiliary definitions 2b⟩
  ⟨Definition of main 1c⟩
  val _ = main ()
end
```
Uses `main` 1c 1c.

We want to do I/O. Use the old `BasicIO` interface because that's what `Lexing` currently hooks up with, unfortunately.

1b ⟨*Modules to open* 1b⟩≡ (1a)
```
BasicIO
Nonstdio
```

`main` parses the command line to determine what input stream to compile from, then spawns off the compile.

1c ⟨*Definition of* `main` 1c⟩≡ (1a)
```
fun main () =
let
  val argv = Mosml.argv ()
  val is = ⟨Open the indicated input stream 2a⟩
  val lexbuf = ⟨Create the lexer stream 2c⟩
  val formatAction = ⟨Parse the lexer stream 2e⟩
  val action = Format.create();
in
  formatAction action
end
```
Defines:
- `action`, never used.
- `argv`, used in chunk 2a.
- `formatAction`, never used.
- `is`, used in chunk 2.
- `lexbuf`, used in chunks 2 and 3.
- `main`, used in chunk 1a.

Determine the input stream to open based on the command line. We will accept either no arguments, indicating standard input is to be read, or one argument, indicating a named file is to be read.

[FMC] We currently have no way of transparently opening up a pipe to the C preprocessor, which was the interface in the original C program.

2a ⟨*Open the indicated input stream* 2a⟩≡ (1c)
```
(case argv of
    [_] => std_in
  | [_, name] => (open_in name
                  handle (SysErr _) =>
                    fatal ("Failed to open " ^ name))
  | arg0::_ => fatal ("Usage: " ^ arg0 ^ " [file]")
)
```
Uses `argv` 1c 1c and `fatal` 2b 2b.

2b ⟨*Auxiliary definitions* 2b⟩≡ (1a) 2d ▷
```
fun fatal s =
(
  output(std_err, s ^ "\n");
  exit 1
)
```
Defines:
`fatal`, used in chunks 2a and 3a.

2c ⟨*Create the lexer stream* 2c⟩≡ (1c)
```
createLexerStream is
```
Uses `createLexerStream` 2d 2d and `is` 1c 1c.

2d ⟨*Auxiliary definitions* 2b⟩+≡ (1a) ◁ 2b 3a ▷
```
fun createLexerStream (is : instream) =
  Lexing.createLexer
  (fn buff => fn n => Nonstdio.buff_input is buff 0 n)
```
Defines:
`createLexerStream`, used in chunk 2c.
Uses `is` 1c 1c.

2e ⟨*Parse the lexer stream* 2e⟩≡ (1c)
```
parseMain Parser.program Lexer.Token lexbuf
```
Uses `lexbuf` 1c 1c and `parseMain` 3b.

We handle a parse error by outputting an error message. There is no attempt at error recovery because `mosmlyac` does not provide convenient support for it. We also catch lexical errors. In either case, we simply die.

3a ⟨*Auxiliary definitions* 2b⟩+≡ (1a) ◁2d 3b▷

```
fun parsePhrase parsingFun lexingFun lexbuf =
  parsingFun lexingFun lexbuf
  handle
    Parsing.ParseError _ =>
      let
        val pos1 = Lexing.getLexemeStart lexbuf
        val pos2 = Lexing.getLexemeEnd lexbuf
      in
        fatal ("Syntax error [" ^
               (Int.toString pos1) ^ ", " ^
               (Int.toString pos2) ^ "]")
      end
  | Lexer.LexicalError (str, num1, num2) =>
    fatal ("Lexer error [" ^
             (Int.toString num1) ^ ", " ^
             (Int.toString num2) ^ "]: " ^
             str)
;
```

Defines:
`parsePhrase`, used in chunk 3b.
Uses `fatal` 2b 2b and `lexbuf` 1c 1c.

This is a wrapper to make sure we clean up the lexer and parser.

3b ⟨*Auxiliary definitions* 2b⟩+≡ (1a) ◁3a

```
fun parseMain parsingFun lexingFun lexbuf =
  let
    val mainPhrase = parsePhrase parsingFun lexingFun lexbuf
      handle x => (Parsing.clearParser(); raise x)
  in
    Parsing.clearParser();
    mainPhrase
  end
;
```

Defines:
`parseMain`, used in chunk 2e.
Uses `lexbuf` 1c 1c and `parsePhrase` 3a.

1 Indices

1.1 Chunks

1.2 Identifiers