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

Franklin Chen

July 28, 1996

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

?? ⟨* ??⟩≡
```
local open ⟨Modules to open ??⟩ in
  ⟨Auxiliary definitions ??⟩
  ⟨Definition of main ??⟩
  val _ = main ()
end
```
Uses `main`.

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

?? ⟨Modules to open ??⟩≡ (? 0—1)
```
BasicIO
Nonstdio
```

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

?? ⟨Definition of main ??⟩≡ (? 0—1)
```
fun main () =
let
  val argv = Mosml.argv ()
  val is = ⟨Open the indicated input stream ??⟩
  val lexbuf = ⟨Create the lexer stream ??⟩
  val formatAction = ⟨Parse the lexer stream ??⟩
  val action = Format.create();
in
  formatAction action
end
```
Defines:
`action`, never used.
`argv`, never used.
`formatAction`, never used.
`is`, never used.
`lexbuf`, never used.
`main`, never used.

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.

?? ⟨*Open the indicated input stream* ??⟩≡ (? 0—1)

```
(case argv of
    [_] => std_in
  | [_, name] => (open_in name
                  handle (SysErr _) =>
                    fatal ("Failed to open " ^ name))
  | arg0::_ => fatal ("Usage: " ^ arg0 ^ " [file]")
)
```

Uses argv and fatal.

?? ⟨*Auxiliary definitions* ??⟩≡ (? 0—1) ??▷

```
fun fatal s =
(
  output(std_err, s ^ "\n");
  exit 1
)
```

Defines:
fatal, never used.

?? ⟨*Create the lexer stream* ??⟩≡ (? 0—1)

```
createLexerStream is
```

Uses createLexerStream and is.

?? ⟨*Auxiliary definitions* ??⟩+≡ (? 0—1) ◁?? ??▷

```
fun createLexerStream (is : instream) =
  Lexing.createLexer
  (fn buff => fn n => Nonstdio.buff_input is buff 0 n)
```

Defines:
createLexerStream, never used.
Uses is.

?? ⟨*Parse the lexer stream* ??⟩≡ (? 0—1)

```
parseMain Parser.program Lexer.Token lexbuf
```

Uses lexbuf and parseMain.

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.

?? ⟨*Auxiliary definitions* ??⟩+≡ (? 0—1) ◁?? ??▷

```
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`, never used.
Uses `fatal` and `lexbuf`.

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

?? ⟨*Auxiliary definitions* ??⟩+≡ (? 0—1) ◁??

```
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`, never used.
Uses `lexbuf` and `parsePhrase`.

1 Indices

1.1 Chunks

1.2 Identifiers