Lexer for Compiler in `mosmllex`

Franklin Chen

July 28, 1996

This is a lexer for the project developed in [?], done in `mosmllex` for Moscow ML rather than in `lex` or `flex` for C.

1 `Lexer` signature

We begin by providing the desired signature.

```
??      ⟨Lexer.sig ??⟩≡
          exception LexicalError of string * int * int;
          val Token : Lexing.lexbuf -> Parser.token;
        Defines:
          LexicalError, never used.
          Token, never used.
```

2 `Lexer` specification

The specification file has a well-defined format:

```
??      ⟨* ??⟩≡
          {
          ⟨Header ??⟩
          }
          ⟨Entry points ??⟩
          ;
```

2.1 Header

```
??      ⟨Header ??⟩≡                                        (? 0—1)
          ⟨Modules to open ??⟩
          ⟨Auxiliary definitions ??⟩
```

We need to access the tokens that `Parser` expects.

```
??      ⟨Modules to open ??⟩≡                               (? 0—1)
          open Parser;
```

We raise `LexicalError` with a message and two character numbers, if we run into a problem while lexing.

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

```
exception LexicalError of string * int * int;

fun lexerError lexbuf s =
  raise LexicalError (s, getLexemeStart lexbuf, getLexemeEnd lexbuf)
;
```

Defines:
`lexerError`, never used.
`LexicalError`, never used.

We maintain a simple symbol table mapping `string`s to reserved word tokens. A hash table implementation from module `Hasht` is used.

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

```
val keyword_table = (Hasht.new 7 : (string,token) Hasht.t);

val () =
List.app (fn (str,tok) => Hasht.insert keyword_table str tok)
[
  ("break", BREAK),
  ("continue", CONTINUE),
  ("else", ELSE),
  ("if", IF),
  ("int", INT),
  ("return", RETURN),
  ("while", WHILE)
];

fun mkKeyword lexbuf =
  let val s = getLexeme lexbuf in
    Hasht.find keyword_table s
    handle Subscript => Identifier s
  end
;
```

Defines:
`keyword_table`, never used.
`mkKeyword`, never used.

2.2 Entry points

Now we present the entry points. For now, we have only one.

?? ⟨*Entry points* ??⟩≡ (? 0—1)

```
rule Token = parse
  ⟨Patterns and actions ??⟩
```

Defines:
`Token`, never used.

?? ⟨*Patterns and actions* ??⟩≡ (? 0—1)

```
    ⟨Preprocessor info ??⟩
  | ⟨Fixed string tokens ??⟩
  | ⟨Keywords and identifiers ??⟩
  | ⟨Constants ??⟩
  | ⟨White space ??⟩
  | ⟨End of file token ??⟩
  | ⟨Illegal token ??⟩
```

[FMC] We still need to properly handle line number adjustments.

?? ⟨*Preprocessor info* ??⟩≡ (? 0—1)

```
  `\n` "#" [`0`-`9`]+ ([` ` `\t`]+ _* )? `\n`
  {
    Token lexbuf
  }
```

Uses Token.

?? ⟨*Fixed string tokens* ??⟩≡ (? 0—1)

```
  ";" { SEMI }
  | "(" { LPAREN }
  | ")" { RPAREN }
  | "{" { LBRACE }
  | "}" { RBRACE }
  | "+" { PLUS }
  | "-" { MINUS }
  | "*" { TIMES }
  | "/" { DIVIDE }
  | "%" { REM }
  | ">" { GT }
  | "<" { LT }
  | ">=" { GE }
  | "<=" { LE }
  | "==" { EQ }
  | "!=" { NE }
  | "&" { AMP }
  | "^" { CARET }
  | "|" { BAR }
  | "=" { ASSIGN }
  | "+=" { PE }
  | "-=" { ME }
  | "*=" { TE }
  | "/=" { DE }
  | "%=" { RE }
  | "++" { PP }
  | "--" { MM }
  | "," { COMMA }
```

Keywords are distinguished from identifiers by means of a lookup into the fixed table of keywords.

?? ⟨*Keywords and identifiers* ??⟩≡ (? 0—1)
```
[`A`-`Z` `a`-`z` `_`] [`A`-`Z` `a`-`z` `_` `0`-`9`]*
{
  mkKeyword lexbuf
}
```
Uses `mkKeyword`.

Only integer constants are currently supported.

?? ⟨*Constants* ??⟩≡ (? 0—1)
```
[`0`-`9`]+
{
  let
    val str = getLexeme lexbuf
  in
    case Int.fromString str of
        NONE => lexerError
                lexbuf
                ("Failed to convert string \"" ^ str ^ "\" to integer")
      | SOME i => Constant i
  end
}
```
Uses `lexerError`.

Skip blanks. [FMC] We really need to keep line number info.

?? ⟨*White space* ??⟩≡ (? 0—1)
```
[` ` `\t` `\n`]+
{
  Token lexbuf
}
```
Uses `Token`.

?? ⟨*End of file token* ??⟩≡ (? 0—1)
```
(eof) { EOF }
```

?? ⟨*Illegal token* ??⟩≡ (? 0—1)
```
_
{
  lexerError
  lexbuf
  ("Illegal token \"" ^
   (getLexeme lexbuf) ^
   "\" in input")
}
```
Uses `lexerError`.

3 Limitations

The # line/file directives, intended to be emitted by a preprocessor, are not lexed correctly in the case of its being the first line of input.

Location information is not currently being maintained at all in the lexer. Line number and column number information should ideally be kept up to date, for the purpose of error messages. The line directives mentioned above should also adjust the line number information (and file information) appropriately.

Anyway, error handling is also nonexistent.

4 Indices

4.1 Chunks

4.2 Identifiers

References

[1] Axel T. Schreiner and H. George Friedman, Jr. *Introduction to Compiler Construction with UNIX*[1] . Prentice-Hall, Inc., New Jersey, 1985.

[1]UNIX is a trademark of Bell Laboratories.