1 Introduction

Your first assignment is to implement a lexer (or scanner) for Mini-Lua, which will convert an input stream of characters into a stream of tokens. While such programs are often best written using a lexer generator (e.g., ML-Lex or Flex), for this assignment you will write a scanner from scratch.

2 Mini-Lua lexical conventions

Mini-Lua has four classes of token: identifiers, delimiters and operators, numbers, and string literals. Tokens can be separated by whitespace and/or comments.

Identifiers in Mini-Lua can be any string of letters, digits, and underscores, not beginning with a digit. Identifiers are case-sensitive (e.g., foo is different from Foo). The following identifiers are reserved as keywords:

```
and break do else elseif end false for function if
in local nil not or repeat return then true until while
```

Note that these are the keywords of Lua; repeat and until are reserved in Mini-Lua, but not used.

Mini-Lua also has a collection of delimiters and operators, which are the following:

```
+ - * / ^ =
~ = <= >= < > ==
( ) { } [ ]
; : , , . . .
```

Numbers in Mini-Lua are integers and are their literals are written using decimal notation (without a sign).

String literals are delimited by matching double quotes and can contain the following C-like escape sequences:
A character in a string literal may also be specified by its numerical value using the escape sequence ‘\ddd,’ where \ddd is a sequence of three decimal digits. Strings in Lua may contain any 8-bit value, including embedded zeros, which can be specified as ‘\000.’

Comments start anywhere outside a string with a double hyphen (--). If the text immediately after -- is different from [[, the comment is a short comment, which runs until the end of the line. Otherwise, it is a long comment, which runs until the corresponding ]]. Long comments may run for several lines and may contain nested [ [/] pairs.

Whitespace is any non-empty sequence of spaces (ASCII code 32), horizontal or vertical tabs, form feeds, newlines, or carriage returns. Any other non-printable character should be treated as an error.

3 Requirements

Your implementation should include (at least) the following two modules:

structure LuaLexer : LUA_LEXER
structure LuaTokens : LUA_TOKENS

The signature of the LuaLexer module is

signature LUA_LEXER =
  sig
    val lexer : ((char, 'a) StringCvt.reader)
               -> (LuaTokens.token, 'a) StringCvt.reader
  end

The StringCvt.reader type is defined in the SML Basis Library as follows:

type ('item, 'strm) reader = 'strm -> ('item * 'strm) option

A reader is a function that takes a stream and returns a pair of the next item and the rest of the stream (it returns NONE when the end of the stream is reached). Thus, lexer is a function that takes a character reader and returns a token reader.

The signature of the LuaTokens module should have the following form:
signature LUA_TOKENS =
  sig
data_type token
    = EOF
    | KW_and
    | KW_break
    | KW_do
    | ...
    | KW_while
    | PLUS | MINUS | TIMES | DIV | EXP | DOTDOT
    | NOTEQ | LTE | GTE | LT | GT | EQEQ
    | EQ | DOT | COLON
    | COMMA | SEMI
    | LP | RP
    | LCB | RCB (* '{' '}' *)
    | LSB | RSB (* '[ ']' *)
    | NAME of Atom.atom
    | NUMBER of IntInf.int
    | STRING of string
end

The EOF token is used to mark the end of stream. The other tokens correspond to the various keywords, delimiters and operators, and literals. The NAME token is for non-reserved identifiers and carries a unique string representation of the identifier. The NUMBER token carries the value of the literal, as does the string representation.