CMSC 22610	**Implementation**	**Project 1**
Winter 2011	**of**	**January 6, 2011**
	Computer Languages I	

MinML lexer

Due: January 21, 2011

1 Introduction

Your first assignment is to implement a lexer (or scanner) for MinML, 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 MinML lexical conventions

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

Identifers can be any string of letters, digits, underscores, and quote marks, beginning with a letter. Identifiers are used to denote type constructors, type variables, data constructors, and value variables. Identifiers are case-sensitive (*e.g.*, `foo` is different from `Foo`). Type constructor and data constructor identifiers must begin with a capital letter, and type variable and value variables must begin with a lower-case letter.

The following identifiers are reserved as keywords:

`and`	**`case`**	**`datatype`**	**`end`**	**`else`**
`fun`	**`fn`**	**`if`**	**`in`**	**`let`**
`of`	**`then`**	**`type`**	**`val`**	

MinML also has a collection of delimiters and operators, which are the following:

`(`	`)`	`==`	`<>`	`<=`	`<`	`>=`	`>`
`~`	`+`	`-`	`*`	`/`	`%`	`||`	`&&`
`@`	`=`	`=>`	`|`	`->`	`,`	`;`	`:`

Numbers in MinML are integers and 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`	—	bell (ASCII code 7)
`\b`	—	backspace (ASCII code 8)
`\f`	—	form feed (ASCII code 12)
`\n`	—	newline (ASCII code 10)
`\r`	—	carriage return (ASCII code 13)
`\t`	—	horizontal tab (ASCII code 8)
`\v`	—	vertical tab (ASCII code 11)
`\\`	—	backslash
`\"`	—	quotation mark

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 MinML may contain any 8-bit value, including embedded zeros, which can be specified as '`\000`.'

Comments start anywhere outside a string with "`(*`" and are terminated with a matching "`*)`". As in SML, comments may be nested.

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 Token
structure MinMLLexer : MinML_LEXER
```

The signature of the `MinMLLexer` module is

```
signature MinML_LEXER =
  sig
    val lexer : ((char, 'a) StringCvt.reader)
          -> (MinMLTokens.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. Notice that the lexer function is polymorphic in the type of the character stream being analyzed. It takes the character reader for the character stream as an argument; this corresponds to the `charRdr` component of the `CHAR_STREAM` signature in the `LexerFn` functor for Fun.

The definition of the `Token` module will be as follows:

```
structure Token =
struct
  datatype token
    = IDU of string      (* identifier, initial upper case *)
    | IDL of string      (* identifier, initial lower case *)
    | INT of int         (* positive integer literal *)
    | STR of string      (* string literal *)
    | NEG                (* integer negation, ~ *)
    | PLUS | MINUS       (* additive operators (+, -) *)
    | TIMES | DIV | MOD  (* multiplicative operators ( *, /, %) *)
    | EQUAL | NOTEQUAL | LESS | GREATER | LESSEQ | GREATEREQ
        (* relational operators (==, <>, <, >, <=, >=)  *)
    | CONCAT             (* string concat (@) *)
    | AND | OR           (* boolean operators (&&, ||) *)
    | FN | DARROW        (* function expression keywords (fn, =>) *)
    | LET | IN | EQ | VAL | FUN | END  (* let, in, =, val, fun, end *)
       (* declaration keywords and punctuation *)
    | IF | THEN | ELSE   (* conditional expr keywords (if, then, else) *)
    | CASE | OF | BAR    (* case expressions (case, of, |) *)
    | TYPE | DATATYPE    (* type declarations (type, datatype) *)
    | TARROW             (* function type arrow (->) *)
    | LPAR | RPAR        (* left and right parentheses ("(", ")") *)
    | COMMA | COLON | SEMI  (* punctuation (",", ":", ";") *)
end  (* structure Token *)
```

The tokens correspond to the various keywords, delimiters, operators, and literals. The `IDU` tokens are initial upper case alphanumeric identifiers representing type constructors and data constructors. The `IDL` tokens are initial lower case alphanumeric identifiers representing type variables and value variables. Either form of identifier token carries a string representation of the identifier.

The `INT` and `STRING` tokens carry the value of the literal.

4 Document History

[1/13/2011] Revised definition of operators and delimiters, and provided a revised version of the Token structure.

[1/14/2011] Removed TRUE and FALSE from token constructors. The boolean constants True and False are just instances of the IDU token class.