CSE341 – Programming Languages (Fall 2014)
Homework #2

Handed out: 3:00pm Tuesday October 28, 2014.

Due: 3:00pm Tuesday November 11, 2014.

Hand-in Policy: Source code should be handed in via Moodle.
Collaboration Policy: No collaboration is permitted. Any cheating (copying someone elses work in any form) will result in a grade of -100 for the first offense and -200 for the subsequent ones.
Grading: Each homework will be graded on the scale 100. Unless otherwise noted, the questions will be weighed equal.

Lexical Analyzer for a Subset of Scheme written in Scheme (100 points): Recall that a lexical analyzer takes a program and generates tokens to be used in parsing. In this homework, you will develop a lexer for a subset of Scheme programming language in Scheme.

A Subset of Scheme Language

Terminal Symbols	**(**, **)**, **integer_literal**, **boolean_literal**, **string_literal**, **quote**, **lambda**, **if**, **let**, **define**, **and**, **or**, **not**, **identifier**
Constructs	**primitive-literal**, **atom**, **list**
	A ***primitive-literal*** is one of the following • an **integer_literal** • a **boolean_literal** • a **string_literal**
	An ***atom*** is one of the following • a ***primitive-literal*** • an **identifier**
	A ***list*** has the form • **(** ***list-items*** **)** where ***list-items*** is a sequence of zero or more of the following (in any combination): • an ***atom*** • a ***list***
Expressions	An ***expression*** is one of the following: • an ***atom*** • a ***list-literal*** • an ***if-expression*** • a ***let-expression*** • a ***lambda-expression*** • a ***function-application***
List literals	A ***list-literal*** has the form

	- (**quote** list-or-atom)
If expressions	An if-expression has the form - (**if** expression expression expression)
Let expressions	A *let-expression* has the form - **(let (** *let-pair-list* **)** *expression* **)** A *let-pair-list* is a sequence of zero or more occurrences of *let-pair*. A *let-pair* has the form - **(identifier** *expression* **)**
Lambda expressions	A *lambda-expression* has the form - **(lambda (** *formals-list* **)** *expression* **)** A *formals-list* is a sequence of zero or more occurrences of **identifier**.
Function application	A function application has the form - **(** *expression arg-list* **)** An *arg-list* is a sequence of zero or more occurrences of *expression*.
Top-level items	A ***top-level-item*** is one of the following: - an **expression** - a ***definition*** A ***definition*** has the form - (**define identifier** expression)
Program	A program is a sequence of one or more occurrences of **top-level-item**.

Project Description

Given the language defined above, implement a lexer that generates the terminal and non-terminal symbols for parsing. The tokens and their corresponding lexeme are given in the following table.

TOKEN	**LEXEME**
LPAREN	“(“
RPAREN	“)”
INTEGER_LITERAL	Any sequence of one or more digits (‘0’,…,’9’)
BOOLEAN_LITERAL	“#t” or “#f”
STRING_LITERAL	Formed by a single double-quote (") character, followed by any sequence of zero or more non-double-quote characters, followed by a single double-quote (") character
QUOTE_KEYWORD	"quote"
AND_KEYWORD	"and"
LAMBDA_KEYWORD	"lambda"
IF_KEYWORD	"if"

DEFINE_KEYWORD	"define"
OR_KEYWORD	"or"
NOT_KEYWORD	"not"
IDENTIFIER	formed by one identifier-character, followed by any sequence of zero or more identifier-character-or-digit characters, where the entire lexeme would not match any other token type. (For example, the lexeme "quote" is a QUOTE_KEYWORD token, not an IDENTIFIER token.)
identifier-character	a letter or any of the following characters: ! $ % & * + - . / : < = > ? @ ^ _ ~
identifier-character-or-digit	a character that is either an itentifier-character or a digit ('0' .. '9')
Note that space characters (space, tab, newline, etc.) are not significant, except when they occur within a string literal.	

A token has two pieces of information:

- the token type: The token type is a member of the TokenType enumeration.
- the lexeme: The lexeme is the token's sequence of characters as they appear in the input file. The lexeme is significant because some kinds of tokens -for example, identifiers- are represented by many possible lexemes. For example, the strings "a", "+", and "eq?" are all identifiers.

Example

Consider the following input text:

```
(define factorial
  (lambda (n)
    (if (= n 1)
        1
        (* n (factorial (- n 1)))
     )
  )
)
```

When reading this input, your lexical analyzer should output the following sequence of tokens:

Token type	Lexeme
LPAREN	(
DEFINE_KEYWORD	define
IDENTIFIER	factorial
LPAREN	(

Token type	Lexeme
LAMBDA_KEYWORD	lambda
LPAREN	(
IDENTIFIER	n
RPAREN	)
LPAREN	(
IF_KEYWORD	if
LPAREN	(
IDENTIFIER	=
IDENTIFIER	n
INTEGER_LITERAL	1
RPAREN	)
INTEGER_LITERAL	1
LPAREN	(
IDENTIFIER	*
IDENTIFIER	n
LPAREN	(
IDENTIFIER	factorial
LPAREN	(
IDENTIFIER	-
IDENTIFIER	n
INTEGER_LITERAL	1
RPAREN	)
RPAREN	)
RPAREN	)
RPAREN	)
RPAREN	)
RPAREN	)

How To Get Started

Copy the files *lexer.ss* and *test.ss* into your own subdirectory. You will implement your lexer in file *lexer.ss*. File *test.ss* contains a few test cases for your convenience. You should use your own test cases as well.

Submission and Grading

You will submit your version of file *lexer.ss* via Moodle.