Project 2: Scheme Lexer and Parser

Contents

1 **Project 2: Scheme Lexer and Parser** 2
- 1.1 *Optional checkpoint due Wednesday, Feb 5, 2025 at 8pm ET* 2
- 1.2 *Final deadline Wednesday, Feb 12, 2025 at 8pm ET* 2
- 1.3 Reported Time to Complete the Project 3

2 **Optional Checkpoint** 3

3 **Background** 4
- 3.1 Lexer 5
  - Lexical Analysis 5
- 3.2 Parser 6
  - Recursive Descent Parsing 6
- 3.3 Scheme I/O Procedures 8
- 3.4 Distribution Code 8
  - Source Files 8
  - Example Files 9
  - Test Harnesses 9
  - Test Files 10
  - Errors 10
  - Token Representation 11

4 **Phase 1: Lexing Individual Token Types** 11
- 4.1 Strings 11
- 4.2 Booleans 12
- 4.3 Characters 12
- 4.4 Numbers 12
- 4.5 Identifiers 12
- 4.6 Punctuators 13
- 4.7 Errors 13
- 4.8 Testing 13

5 **Phase 2: Full Lexer** 13

6 **Phase 3: Parser** 14
- 6.1 Simple Expressions 14
- 6.2 Compound Expressions 14
- 6.3 Errors 15
- 6.4 Testing 15

7 **Rules and Regulations** 16

8 **Grading** 17
- 8.1 Test Grading 18
  - Running Your Own Test Cases 18

9 Submission 19

10 Frequently Asked Questions 19

https://eecs390.github.io/project-scheme-parser/

1 Project 2: Scheme Lexer and Parser

1.1 *Optional checkpoint due Wednesday, Feb 5, 2025 at 8pm ET*

1.2 *Final deadline Wednesday, Feb 12, 2025 at 8pm ET*

In this project, you will implement a lexer and parser for the R5RS Scheme programming language. The main purpose of this exercise is to gain experience in functional programming and Scheme. You will also get practice reasoning about the lexical and syntactic structure of a language.

This project must be written in R5RS-compliant Scheme. The officially supported interpreter for this project is Racket[1]. Make sure you choose to run R5RS Scheme. If you use the DrRacket interface, select *Language* -> *Choose Language* -> *Other Languages* -> *R5RS* from the menu. You may need to click on *Run* before the interface will show that R5RS is chosen.

By default, the `Makefile` in the distribution code is set up to use the `plt-r5rs` command-line interpreter included in the Racket distribution. You may need to add the `bin` directory under your Racket installation to your path[2] so that the `plt-r5rs` executable can be located. You may also modify the `SCHEME` variable in the `Makefile` to point at the command-line interpreter you wish to use.

The project is divided into multiple suggested phases. We recommend completing the project in the order of the phases below.

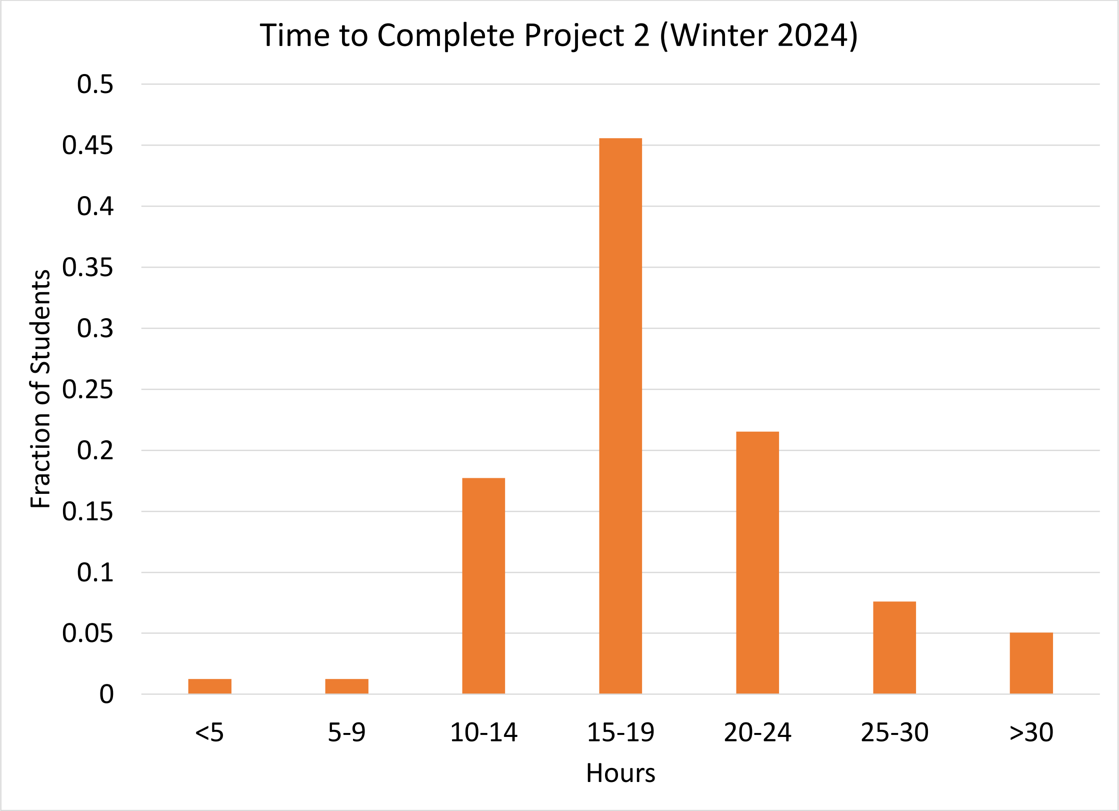
You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes both code and test cases.

[1] On MacOS, you can install Racket with Homebrew (`brew install --cask racket`).

[2] Instructions: Windows; MacOS (e.g. `export PATH="/Applications/Racket v7.4/bin:$PATH"` for a Racket 7.4 installation on MacOS; this is unnecessary if you installed Racket with Homebrew)

1.3 Reported Time to Complete the Project

The following is the time students reported they spent on the project in Winter 2024.



These data are for planning purposes only. We do not consider the exact time spent on the project to be reflective of anyone's learning or ability. Rather, completing the project regardless of how much time it takes is what is important to achieve the learning goals of the project.

2 Optional Checkpoint

The checkpoint consists of achieving at least 30% of the points on the public and private test cases before the checkpoint deadline, which can be achieved by passing all the public tests for *Phase 1* (including error checking). Your grade for the checkpoint will be computed as the better of:

- $min(0.3, score)/0.3$, where $score$ is the fraction of autograded points earned by your best submission before the checkpoint deadline.
- $finalScore$, where $finalScore$ is the fraction of autograded points earned by your best submission before the final deadline.

Thus, completing the checkpoint is **optional**. However, doing it will work to your benefit, since you can guarantee full credit on the 25% of the project points dedicated to the checkpoint.

For this project, passing the public test cases for Phase 1 is enough to guarantee full credit on the checkpoint.

3 Background

The input to an interpreter is raw character data from an input stream. For instance, when a user types in `(+ 1 3.1)` into an interactive Scheme interpreter, the interpreter receives the raw character sequence:

```
#\( #\+ #\space #\1 #\space #\3 #\. #\1 #\)
```

This character sequence is called the *external representation* of an object (Section 3.3 in the R5RS spec). Before it can perform any evaluation, the interpreter must first parse the raw character data into a structured *internal representation* that is easier to reason about. In Scheme, the `read` procedure takes the character data from an input stream and produces a Scheme-level representation: for `(+ 1 3.1)`, the internal representation is a Scheme list whose elements are the symbol `+`, the integer `1`, and the real number `3.1`. The `eval` procedure can then be used to evaluate the Scheme representation in a given environment.

External vs. Internal Representation

The external representation of an object is how it is represented as a character sequence in source code, while the internal representation is its in-memory representation in a compiler, interpreter, or runtime. In C++, for instance, the source-code representation of the literal `3.14f` consists of the five-character ASCII sequence `'3'`, `'.'`, `'1'`, `'4'`, `'f'`. The runtime representation in most implementations is the binary value `01000000010010001111010111000011`, which is the 32-bit IEEE-754 representation of the closest value to 3.14 that can be represented in that format. Observe that the source-code representation consists of decimal digits, a decimal point, and the float specifier `f`, all encoded in ASCII, while the runtime representation consists of just bits. It is the lexer and parser that convert between these representations (either directly or indirectly through an intermediate, compile-time representation).

The external representation of an object is not necessarily unique. For example, the literals `3.`, `3.0`, and `3.00` all represent the same floating-point value, with a common internal representation. As another example, the sequences `'x` and `(quote x)` are both external representations in Scheme for a list that contains the `quote` symbol as its first element and the symbol `x` as its second element. When printing out a particular object, an implementation chooses one external representation to use, which may be different than the one that the programmer used.

Finally, the internal representation of an expression is distinct from the value to which it evaluates. In Scheme, the internal representation for the expression `(+ 1 3)` consists of a list that contains the `+` symbol, the number `1`, and the number `3`. When evaluated, the expression results in the value `4`, which is nowhere to be seen in the internal representation. Another example is the expression `x`, whose internal representation is just the symbol `x`. When evaluated, this expression can result in any number of values depending on what `x` is bound to in the current environment, or even in an error if no binding for `x` exists in the environment.

The job of the lexer and parser is to produce the internal representation of a code fragment. They do not evaluate or execute the fragment.

The following examples in an interactive interpreter illustrate the `read` and `eval` procedures. The input to each `read` call is provided immediately after the expression in which `read` appears:

```
> (read) (+ 1 3.1)      ; input is (+ 1 3.1)
(+ 1 3.1)
> (equal? (read) (list '+ 1 3.1)) (+ 1 3.1)
#t
> (eval (read) (scheme-report-environment 5)) (+ 1 3.1)
4.1
```

In this project, you will implement the `read-datum` procedure, which performs the same computation as the built-in `read` procedure. In *Phase 1* and *Phase 2*, you will implement a *lexer*, which translates raw character data into tokens.

For example, the input `(+ 1 3.1)` will result in the following tokens:

```
(punctuator "(")
(identifier +)
(number 1)
(number 3.1)
(punctuator ")")
```

You will then implement a *parser* in *Phase 3*, which translates tokens into structured Scheme-level representations that can then be evaluated or manipulated as data.

Start by familiarizing yourself with the lexical structure of Scheme, as defined in Sections 2 and 7.1.1 of the R5RS spec, and the grammatical structure as defined in Section 7.1.2.

3.1 Lexer

Recall that the lexical structure of a language determines what constitute the *tokens* of the language, which are akin to the words in a natural language. The lexical structure can be expressed with regular expressions, though some languages use the same Backus-Naur form (BNF) to specify both the lexical and grammatical structure. The Scheme R5RS spec follows the latter strategy, defining the lexical structure in Section 7.1.1 of the spec.

In this project, we will be handling most of the standard R5RS lexical specification. However, we depart from the specification in the following:

- Our lexer will not distinguish between keywords and other identifiers. Thus, the nonterminals $\langle\text{syntactic keyword}\rangle$, $\langle\text{expression keyword}\rangle$, and $\langle\text{variable}\rangle$ do not appear in our lexical specification. Instead, all three categories are lexed as $\langle\text{identifier}\rangle$.
- We restrict numbers to base-10 decimal and integer literals. We do not handle other bases, complex numbers, or fractions. We also restrict the allowed decimal formats. The following replaces the specification for the $\langle\text{number}\rangle$ nonterminal:

$$\begin{aligned}
\langle\text{number}\rangle &\longrightarrow \langle\text{integer}\rangle \mid \langle\text{decimal}\rangle \\
\langle\text{integer}\rangle &\longrightarrow \langle\text{sign}\rangle\ \langle\text{digit}\rangle^{+} \\
\langle\text{decimal}\rangle &\longrightarrow \langle\text{sign}\rangle\ \langle\text{digit}\rangle^{+}\ .\ \langle\text{digit}\rangle^{*} \mid \langle\text{sign}\rangle\ .\ \langle\text{digit}\rangle^{+}
\end{aligned}$$

We will take a narrow interpretation of the rules in the R5RS spec. In particular, the spec states:

> Tokens which require implicit termination (identifiers, numbers, characters, and dot) may be terminated by any $\langle\text{delimiter}\rangle$, but not necessarily by anything else.

We will enforce this rigidly. Thus, inputs such as `asdf,` (including the comma) and `3a` are erroroneous, rather than being lexed as identifiers as in some Scheme implementations.

Lexical Analysis

Lexical analysis is generally implemented with a finite-state machine (FSM). An FSM consists of a fixed number of states, and transitions are made between states upon an incremental input, usually individual characters or a character sequence in the case of a lexer. The *start* state is the initial state of the machine, and *accepting* states are those that indicate that the given input is valid.

An FSM is often illustrated as a directed graph, with a node for each state and accepting states denoted by a doubled node boundary. Edges correspond to transitions between states and are labeled by the incremental input that triggers the transition. As an example, Figure 1 is an FSM for lexing strings in Scheme.

A double-quotation marker causes a transition from the start state to the `string*` state. Any subsequent character results in the same state, unless it is a backslash or double-quotation marker. The former indicates the start of an escaped

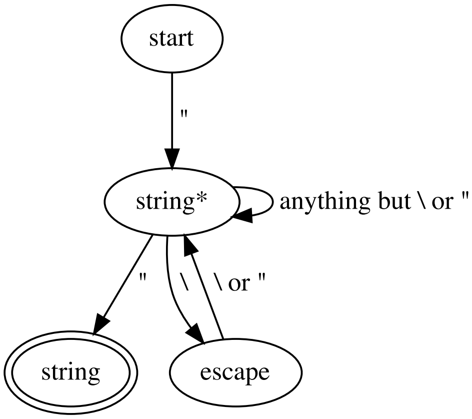


Figure 1: A finite-state machine for lexing a Scheme string.

backslash or double-quotation marker, so it causes a transition to the `escape` state. A backslash or double-quotation returns back to the `string*` state. Any other character results in an error. From the `string*` state, a double-quotation mark indicates the end of the string, causing a transition to the `string` state. This is an accepting state, so the input so far is accepted as a valid string.

An FSM can be implemented as a functional program, with a function for each state. Upon reading a character or sequence of characters that trigger a transition, the function for the new state is invoked. The function for an accepting state merely returns a representation of the accepted token.

For this project, we recommend working out state machines for the Scheme lexical specification. Then use the resulting FSMs to derive the procedural structure of your lexical analyzer.

3.2 Parser

Once the input has been split up into tokens, the tokens must be parsed in order to produce the data that they represent. We will be parsing the *external representation* of Scheme expressions, as described in Sections 3.3 and 7.1.2 of the R5RS spec. This matches what is produced by the built-in `read` procedure.

Recursive Descent Parsing

The algorithm our parser uses is recursive descent parsing. In this algorithm, a nonterminal of the grammar is generally implemented as a function, which recursively calls the functions for the nonterminals that appear on the right-hand side of the rules for the nonterminal. The specific function to call is determined by the tokens that are read. (Tokens are the terminals of the grammar.)

As an example, consider the grammar that corresponds to words that have some number of a's, followed by any number of c's, followed by the same number of b's as a's. Thus, `ab`, `c`, and `aacccbb` are all accepted by the grammar. The grammar specification is as follows, with `S` the start symbol:

$$S \to a\,S\,b \mid C$$
$$C \to c\,C \mid \varepsilon$$

The following is a recursive descent parser for this grammar, with the characters `a`, `b`, and `c` as terminals, implemented in Scheme:

```
; Parses the S nonterminal. Returns a list of two numbers, the first
; the number of a's and b's and the second the number of c's.
(define (parse-S)
  (let ((item (peek-char)))
    (cond ((char=? item #\a) ; next item is an a
           (read-char) ; remove the a from the input stream
           (let* ((recurse (parse-S)) ; recursively parse S
                  (next (read-char))) ; next item should be b
             (if (char=? next #\b)
                 ; add 1 to the a count, preserve c count
                 (cons (+ 1 (car recurse)) (cdr recurse))
                 (error "expected b")
             )
           ))
          (else (list 0 (parse-C))) ; 0 a's + however many c's
    )
  )
)

; Parses the C terminal. Returns the number of c's.
(define (parse-C)
  (let ((item (peek-char)))
    (cond ((char=? item #\c) ; next item is a c
           (read-char) ; remove the c from the input stream
           (+ 1 (parse-C))) ; add 1 to the recursive c count
          (else 0) ; no c's
    )
  )
)
```

The parse-S procedure parses the S nonterminal. If it encounters an a, then the first rule for S must apply, namely

$$S \to a\ S\ b$$

Thus, it removes the a from the input, recursively parses an S, and then requires a b to follow, which is also removed from the input. (We use a let* in the code rather than let since the former guarantees that the initializers run in order, while the latter does not.) If, on the other hand, something other than an a is encountered, the second rule for S must apply, and the procedure instead calls parse-C to parse the C nonterminal.

Similarly, if parse-C encounters a c, then the first rule for C applies and it removes the c and recursively calls parse-C. Otherwise, the second rule applies, which matches C to empty, so that nothing is removed from the input stream.

In the example above, both the parse-S and parse-C functions are recursive, calling themselves. In general, the functions in a recursive descent parser can also be mutually recursive, as will be the case in the Scheme parser.

3.3 Scheme I/O Procedures

In this project, you will be processing data from standard input. You may use the following Scheme procedures in order to read input and write output:

- `read-char`: reads a single character from standard input, removing it from the input stream
- `peek-char`: reads a single character from standard input without removing it from the input stream
- `eof-object?`: determines if the value read from an input stream is the end-of-file object
- `display`: writes a string to standard output; does not write a newline
- `newline`: writes a newline to standard output

You may use any non-mutating standard procedures you like, as long as they are permitted under the *Rules and Regulations*. The following are some procedures that you may find useful:

- `char=?`, `char<=?`, `char>=?`: compare two characters; do not use `eq?`, as its behavior on characters is implementation-dependent
- `list->string`: converts a list of characters into a string
- `string->list`: converts a string into a list of characters
- `string->number`: converts a string representation of a number into its corresponding numerical value
- `string->symbol`: converts a string into a symbol

Refer to the R5RS spec for the full set of available standard procedures.

You may **not** use the built-in `read` procedure, or any other built-in procedure for parsing or evaluating Scheme, except in your test cases. **Submissions that use `read` in the lexer or parser will receive no credit.** You may **not** use any constructs for iteration – use recursion instead. See *Rules and Regulations* for the full set of rules.

3.4 Distribution Code

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/project-scheme-parser/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

Source Files

`distribution.scm`	Utility functions for use in the project. Read through the documentation here, as you will find many of the functions useful.
`lexer.scm`	Starter code for the Scheme lexer. Comments indicate where you will need to add or modify code.
`parser.scm`	Starter code for the Scheme parser.

You will only be modifying `lexer.scm` and `parser.scm`.

Example Files

`simple.scm`	A simple parser for the language described in *Parser*.

Test Harnesses

`util.scm`	Utility functions used in test code.
`token-types-teste scm`	Testing framework for lexing individual token types. Read the documentation at the top of the file to determine how to use the framework.
`tokenize-tester.scm`	Testing framework for lexing an entire Scheme file.
`read-simple=teste scm`	Testing framework for lexing and parsing simple Scheme expressions.
`read-compound-tes scm`	Testing framework for lexing and parsing compound Scheme expressions.
`read-datum-tester scm`	Testing framework for lexing and parsing all Scheme expressions.
`parse-tester.scm`	Testing framework for lexing and parsing an entire Scheme file.
`repl.scm`	A simple read-eval-print loop that uses `read-datum` for parsing and `eval` for evaluation.
`rpl.scm`	A simple read-print loop that uses `read-datum` for parsing and prints out the resulting datums, without evaluating them.
`Makefile`	A Makefile for running test cases.

When writing tests that use these harnesses, read the documentation at the top of the testing harness to see its expected input format. The test drivers (`token-types-tester.scm`, `tokenize-tester.scm`, `read-simple-tester.scm`, `read-compound-tester.scm`, `read-datum-tester.scm`, and `parse-tester.scm`) all read input from standard in and write to standard out. You can use file redirection from the command line to work with input and output files:

```
$ plt-r5rs token-types-tester.scm < string-tokens.in > string-tokens.out
```

Test Files

*tokens.in	Test cases for use with token-types-tester.scm. You can run the tests with make test-tokens.
*tokens.err	Test cases for detecting errors in lexing. You can run the tests with make error-tokens.
*tokens.expect	Expected output for the *tokens.in and *tokens.err tests.
distribution.tokens.expect	Expected output for running tokenize-tester.scm with distribution.scm. You can use make test-tokenize to run the test.
simple.in	Test case for use with read-simple-tester.scm. You can run it with make simple.
simple.expect	Expected output for simple.in.
compound.in	Test case for use with read-compound-tester.scm. You can run it with make compound.
compound.expect	Expected output for compound.in.
datum.in	Test case for use with read-datum-tester.scm. You can run it with make datum.
datum.expect	Expected output for datum.in.
*datum.err	Test cases for detecting errors in parsing. You can run the tests with make error-datum.
*datum.expect	Expected output for the *datum.err tests.
distribution.datums.expect	Expected output for running parse-tester.scm with distribution.scm. You can use make test-parse to run the test.
public_tests.scm	Collected test cases from *.in and *.err files for use with tokenize-tester.scm and parse-tester.scm.
public_tests.tokens.expect	Expected output for running tokenize-tester.scm with public_tests.scm.
public_tests.datums.expect	Expected output for running parse-tester.scm with public_tests.scm.

The * in the filenames above represents a wildcard pattern. For example, *datum.err matches the given files error7-datum.err and error8-datum.err. You should write your own test files that match these patterns, and they will be run when invoking the corresponding make target.

Errors

When your lexer or parser encounters improperly formatted input, you should signal an error by calling the error procedure, defined in distribution.scm, with an appropriate error message. For example, the input asdf, (including the comma) is not a valid identifier, since it is not terminated by a delimiter. Thus, your lexer should invoke the error procedure, as in the following:

```
(error "bad identifier")
```

This aborts lexing or parsing and prints out the error message:

```
Error: bad identifier
```

Token Representation

The output of lexical analysis is a sequence of tokens. In this project, a token is represented as a list of two elements. The first element identifies the type of the token, while the second element is a representation of the data value for the token. In our lexer, the type must be one of the Scheme symbols identifier, boolean, number, character, string, or punctuator. The latter is the category we use for parentheses, the token representing the start of a vector (#(), the dot (.), and Scheme quotation markers (' or ` or , or ,@). (Thus, any token that is not an identifier, boolean, number, character, or string is a punctuator.) Within a token, the data value is represented as follows for each category:

- identifier: a Scheme symbol representing the identifier. For example, reading the input aloha should produce the token (identifier aloha).
- boolean: a Scheme boolean representing the boolean literal. For example, reading the input #f should produce the token (boolean #f).
- number: a Scheme number representing the number literal. For example, reading the input +3.14 should produce the token (number 3.14).
- character: a Scheme character representing the character literal. For example, reading the input #\a should produce the token (character #\a).
- string: a Scheme string representing the string literal. For example, reading the input "hello world" should produce the token (string "hello world").
- punctuator: a Scheme string representing the punctuator (i.e. "(" or ")" or "#(" or "." or "'" or "`" or "," or ",@"). For example, reading the input ,@ should produce the token (punctuator ",@").

The constructor and accessors for the token ADT are defined in the distribution code: token-make, token-type, and token-data. Respect the ADT interface: do not use list manipulators in order to work with tokens.

4 Phase 1: Lexing Individual Token Types

In this phase, we will write separate functions to lex each token type. Each such function is a simpler task then lexing all of Scheme, since it can assume the type of token it is parsing.

4.1 Strings

Start by reading over the starter code for read-string. After discarding whitespace and comments, it checks to make sure that the initial character is a double quote, raising an error if it is not. (The read-start procedure in distribution.scm raises an error if the character it reads does not match the one that is expected.) It then calls the read-string-tail helper function, which will collect the string characters in reverse order in the read-so-far list. The helper function uses get-non-eof-char, another procedure defined in distribution.scm, to read the next character and make sure that it is not an end-of-file character. If the character is a double quote, the string is complete and a string token must be constructed. The read-so-far list is reversed and converted to a string using the built-in list->string procedure. Then token-make is used to encode the token type and data together.

If a backslash is encountered, the read-escaped function is called to read the rest of the escape sequence. This function ensures that the escape sequence is one of those permitted by Scheme, and it returns the actual escaped character itself. This is prepended to the read-so-far list by read-string-tail, which makes a recursive call to read the rest of the string.

The last case, which you will need to complete, is when the character read by read-string-tail is neither the double quote nor the backslash.

You can test your code using the token-types-tester.scm test driver. Some tests are contained in the file string-tokens.in, with the expected output in string-tokens.expect:

```
$ plt-r5rs token-types-tester.scm < string-tokens.in > string-tokens.out
$ diff string-tokens.out string-tokens.expect
```

4.2 Booleans

Complete the `read-boolean-tail` procedure, which reads the tail end of a boolean literal and returns the token representation. Refer to the R5RS spec for what are valid boolean literals. Raise an error if any other data is read.

A handful of test cases are located in `boolean-tokens.in`.

4.3 Characters

Write the `read-character-tail` procedure, which reads all but the start of a character literal. Enforce the requirement that a character literal be terminated by a delimiter; raise an error if this is not the case. (You will find the `delimiter?` predicate in `distribution.scm` useful.) You should ensure that the delimiter remains in the input stream, since it can constitute part of the next token in the stream. (Thus, use `peek-char` rather than `read-char` when you may be reading a delimiter.) Also make sure to properly handle the `#\space` and `#\newline` literals. A few tests are in `character-tokens.in`.

4.4 Numbers

The `read-number` procedure should lex a number literal. As mentioned in *Lexer*, we only handle a restricted set of number formats. Even within this set, however, a number can begin with a sign, a digit, or a dot. As with characters, enforce the requirement that a number be terminated by a delimiter. Make sure to handle decimals properly: raise an error if a number literal contains more than one decimal point.

We recommend drawing out a finite-state machine to work out what helper functions to write and when to call them. Use the FSM for strings in *Lexical Analysis* as a model.

You will likely find the built-in `string->number` procedure useful in constructing the token representation.

The file `number-tokens.in` contains some tests.

4.5 Identifiers

Read through the lexical specification for identifiers in the R5RS spec carefully, as it includes many special cases. As mentioned previously, we will treat keywords as identifiers here. Enforce the requirement that an identifier be terminated by a delimiter.

You must handle upper-case letters, but since case is insignificant, convert them to lower-case letters while lexing. Thus, reading in `Hello` from input should produce the token `(identifier hello)`. You will find the built-in `char-downcase` procedure helpful.

A sign (i.e. + or -) followed by a delimiter denotes an identifier, but if anything other than a delimiter follows the sign, then it cannot be the start of an identifier.

The ellipsis (`...`) is its own special case of an identifier. You must properly handle the fact that three consecutive dots, terminated by a delimiter, is an identifier, but any other number of dots (e.g. `..` or `....`) is not.

As before, we recommend drawing out an FSM to work out the functions you need.

The built-in `string->symbol` procedure will be helpful in constructing the token representation.

A small number of test cases are in `identifier-tokens.in`.

4.6 Punctuators

Implement the read-punctuator procedure to lex a punctuator (i.e. one of ( ) #(. ' ` , ,@). A comma followed by an at (@) symbol is a single punctuator, while a comma followed by anything else indicates just the comma punctuator itself. As discussed in *Token Representation*, use a string representation of the punctuator when constructing the resulting token. Tests are in punctuator-tokens.in.

4.7 Errors

Your lexer should indicate errors by invoking the error procedure. A few test cases for errors are provided in the *tokens.err files.

4.8 Testing

In addition to testing with the token-types-tester.scm harness, you may find it useful to test the lexing procedures interactively. After starting the plt-r5rs interpreter and loading lexer.scm, you can call a lexing function and provide the required input after the call:

```
> (load "lexer.scm")
> (read-string)
"Hello world!"
(string "Hello world!")
> (read-character)
#\space
(character #\space)
> (read-number)
+3.14
(number 3.14)
```

You must also write your own test files, as the given test files only cover a small number of cases. Make use of the provided test harnesses and name your test files according to the same pattern as the provided files, so that they automatically get run by the Makefile as described in *Test Files*.

5 Phase 2: Full Lexer

The next step is to write a single function that can lex any supported Scheme token. Complete the read-token procedure so that it does so.

In some cases, you will find that looking at a single character is not enough to determine what kind of token it is. Examples include a dot (.), which may be a punctuator, part of a number, or part of an identifier, and a hash (#), which may be part of a punctuator, character, or boolean. We suggest drawing out a full state machine that handles any valid input for our subset of Scheme, and then structure your functions accordingly. You will not be able to directly call the functions you wrote in Phase 1 for these cases; those functions assume that the entire token is still in the input stream, but you will have removed at least one character from the input to examine a second character.

Do not unnecessarily repeat code. If you find that you need to repeat code that you've already written as part of the previous phase, restructure your program so that the shared code is in its own helper function that can be called from wherever you need it.

Once you complete read-token, the provided tokenize procedure will lex all of standard input, producing a list of tokens. The test driver tokenize-tester.scm calls this function and writes the result to standard output:

```
$ plt-r5rs tokenize-tester.scm < distribution.scm > distribution.tokens.out
$ diff distribution.tokens.out distribution.tokens.expect
```

6 Phase 3: Parser

Our Scheme parser reads tokens from standard input using the `read-token` function and then parses the tokens in order to produce Scheme data that represent the input expressions. The parser should only consume the tokens needed to build an expression. Thus, you should use `read-token` rather than the `tokenize` procedure.

6.1 Simple Expressions

Complete the `read-simple-datum` procedure, which reads and parses a single simple expression. Recall the *Token Representation*, which consists of the type of the token and a Scheme representation of the data. You should not have to do any further processing on the data contained within a token.

The file `simple.in` contains a few test cases for reading simple data. The `read-simple-tester.scm` test driver compares the output of your parser with that of the built-in `read` procedure:

```
$ plt-r5rs read-simple-tester.scm < simple.in > simple.out
$ diff simple.out simple.expect
```

6.2 Compound Expressions

Compound expressions consist of lists (including dotted lists), vectors, and abbreviations. Fill in the `read-compound-data-helper` procedure, which is used by `read-compound-datum` to read and parse a compound expression.

Lists of the non-dotted variety and vectors consist of an arbitrary sequence of expressions, terminated by a closing parenthesis. You will need to recursively call the `read-datum` or `read-datum-helper` procedures in order to read and parse each of these expressions. You will need to complete the latter procedure before these recursive calls function properly. Do so as part of writing `read-compound-data-helper`.

Pay careful attention to the format of a dotted list: at least one datum must precede the dot, and exactly one must follow between the dot and the closing parenthesis. Raise an error if either condition is violated.

You will need to combine the data in a list or vector into the appropriate data structure. Thus, parsing the input `(1 2)` should produce an actual list containing the elements 1 and 2. You may find the `list->vector` procedure useful in producing the result of parsing a vector.

An abbreviation consists of an abbreviation marker followed by a datum. You will need to turn an abbreviation into its full form, which is a list consisting of the corresponding keyword and the datum. Some examples:

```
'hello --> (quote hello)
`world --> (quasiquote world)
,(a b) --> (unquote (a b))
,@(c d) --> (unquote-splicing (c d))
```

We suggest diagramming your program structure for parsing compound expressions before writing any code. Refer to the simple parser in *Parser* as an example. It may be helpful to work through some examples of compound Scheme expressions to get an idea of how to handle them in your parser. (We do not suggest drawing a finite-state machine here, as an FSM only works for regular expressions and is not powerful enough to recognize even the relatively simple syntactic structure of Scheme.) Use helper functions where appropriate: do not place your code entirely in `read-compound-data-helper` or `read-datum-helper`.

A handful of test cases are in `compound.in` and `datum.in`, and you can use the test drivers `read-compound-tester.scm` and `read-datum-tester.scm` to run the tests. You can also use the `Makefile` to run all the tests provided in the distribution code.

6.3 Errors

As with the lexer, signal errors using the `error` procedure. A few test cases that have errors are provided in the `*datum.err` files, and they can be run with `read-datum-tester.scm`.

6.4 Testing

Once you complete the parser, you can test your code against the built-in `read` procedure: `read-datum` when reading and parsing the same input should produce a result that compares `equal?` with that of `read`. The test drivers `read-simple-tester.scm`, `read-compound-tester.scm`, and `read-datum-tester.scm` work on input files where every expression is repeated twice, using `read` to read the first and `read-datum` for the second. They then compare the results to make sure they are equal.

You may also find it useful to examine the results of `read-datum` and `read` interactively. You can do so by starting the `plt-r5rs` interpreter and loading `parser.scm`:

```
> (load "parser.scm")
> (read)
'(hello world) ; typed, result is on next line
(quote (hello world))
> (read-datum)
'(hello world) ; typed, result is on next line
(quote (hello world))
```

You must write your own test files, as the given test files only cover a small number of cases. As with the lexer, make use of the provided test harnesses and file-naming conventions, as described in *Test Files*.

The test harness `parse-tester.scm` uses `read-datum` to lex and parse all of standard input, writing the parsed data to standard output:

```
$ plt-r5rs parse-tester.scm < distribution.scm > distribution.datums.out
$ diff distribution.datums.out distribution.datums.expect
```

The file `repl.scm` implements a simple *read-eval-print loop (REPL)* that interactively reads in expressions using `read-datum`, evaluates them with the built-in `eval`, and then prints the result. The following is an example of running `repl.scm`:

```
$ plt-r5rs repl.scm
scm> (+ 1 3.1)
4.1
scm> (define x 3)
scm> x
3
scm> (define (hello) (display "Hello world!") (newline))
scm> (hello)
```

(continues on next page)

(continued from previous page)

```
Hello world!
scm>
```

An end-of-file will exit the program. (If you want to get meta, run `plt-r5rs repl.scm` and then copy and paste the contents of `repl.scm` to standard input. You now have a REPL within a REPL within the `plt-r5rs` interpreter! You will have to enter two end-of-files to exit, one for each REPL.)

The file `rpl.scm` implements a read-print loop, without evaluation. It interactively reads in datums using `read-datum`, printing out what was read. The following is an example of running `rpl.scm`:

```
$ plt-r5rs rpl.scm
read> 3
3
read> "\a"
Error: unrecognized escape sequence
read> (+ 1
4)
(+ 1 4)
read> (1 . 2)
(1 . 2)
read> (1 . 2 3)
Error: expected closing parenthesis
read> '3
(quote 3)
read> '''x
(quote (quote (quote x)))
```

An end-of-file will exit the program.

7 Rules and Regulations

The goals of this project are to better understand lexing and parsing, as well as to get experience writing Scheme and functional code. As such, your code is subject to the following constraints:

- You may not use any non-R5RS-standard code, nor external code or code generated by an external library (e.g. by a lexer or parser generator).
- You may not use the built-in `read` or `eval` procedures, or similar procedures, except in testing.
- You may not use any procedures or constructs with side effects, except the I/O procedures `read-char`, `peek-char`, `display`, `write`, and `newline` (you likely won't need the latter three yourself, but they are used in the distribution code). Included in the prohibited set are any mutators (procedures or constructs that end with a bang, e.g. `set!`), and you may only use `define` at the top level (i.e. at global scope). The `read-char` and `peek-char` procedures may only be used in the lexer.
- You may not use any iteration constructs (including the "named `let`" construct described in section 4.2.4 of the R5RS spec). Use recursion instead.

To facilitate checking the rules above, the following symbols may not appear in `lexer.scm` or `parser.scm`:

- `read`
- `eval`
- `current-input-port`

- `current-output-port`
- `open-input-file`
- `open-output-file`
- `with-input-from-file`
- `with-output-to-file`
- any symbol ending with `!`
- `define`, except as the first item in a top-level list
- `peek-char`, except in `lexer.scm`
- `read-char`, except in `lexer.scm`
- `do`
- `for-each`
- `syntax-rules`

Any violations of the two sets of rules above will result in a score of 0.

In addition, the standard Engineering Honor Code rules apply. Thus, you may not share any artifact you produce for this project outside of your partnership, including code, test cases, and diagrams (e.g. of state machines). This restriction continues to apply even after you leave the course. Violations will be reported to the Honor Council.

The functions you write in this project do **not** have to be tail recursive.

8 Grading

The grade breakdown for this project is as follows:

- 25% checkpoint
- 70% final deadline autograded
- 5% final deadline hand graded

Hand grading will evaluate your programming practices, such as avoiding unnecessary repetition and respecting ADT interfaces. We will look for the following specific pitfalls:

- not respecting the token ADT – only `token-make`, `token-type`, and `token-data` should be used to interact with the ADT
- not respecting the interface of the lexer in the parser – the parser should only use `read-token` and no other input procedures
- significant code duplication
- using a conditional to check for set membership rather than a built-in procedure such as `member` or `memv`
- non-descriptive variable, function, or class names
- avoiding the recursive leap of faith, leading to unnecessary cases in recursive functions
- overly lengthy or nested code, where helper functions should be used instead
- unnecessary use of `let` (e.g. `(let ((variable expression)) variable)` instead of just `expression`, or using nested `let`s where a `let*` could be used instead)
- unnecessary conditional or boolean logic (e.g. comparing to `#t` or `#f`, patterns such as “if test then true else false” instead of using the truth value of “test” directly, and so on),

- using nested conditionals where a cond could be used instead
- outdated comments or commented-out code, which make the code harder to read, understand, and maintain
- missing or uninformative documentation

Use the above as a checklist for making sure that your code meets the coding-practices requirements.

To be eligible for hand grading, your solution must achieve at least half the points on the autograded, final-deadline portion of the project.

8.1 Test Grading

We will autograde your tests for *Phase 2* and *Phase 3*, which should be written as plain Scheme input. Since we are only doing syntactic analysis, your test cases need only be concerned with syntax. They do not need to be semantically correct Scheme code.

Your test files must use the following naming pattern:

- `test*.scm`

This pattern does not match any of the public test files – the buggy instructor solutions have been carefully crafted to pass all the public tests, so you will need to write your own tests that provide coverage distinct from the public tests.

You can submit up to 30 files matching the pattern above.

To grade your tests, we use a set of intentionally buggy instructor solutions (or *mutants*).

1. We run your tests with correct solutions for both the lexer and parser, generating corresponding `.expect` files:

   ```
   $ plt-r5rs tokenize-tester.scm < test-file.scm > test-file.tokens.expect
   $ plt-r5rs parse-tester.scm < test-file.scm > test-file.datums.expect
   ```

2. We run all of your tests against each buggy solution, using the `.expect` files generated by the correct solution.

   - If any of your tests fail on the buggy solution, they have caught the bug.
   - The autograder assigns points for each bug that is caught.

You may include any number of erroneous or correct tokens and datums in a single `test*.scm` file. However, keep the following in mind:

- When the lexer or parser encounters an error, it discards the rest of the current line. Therefore, do not include multiple errors on the same line.

Running Your Own Test Cases

The `Makefile` will automatically run your `test*.scm` test cases through `tokenize-tester.scm` and `parse-tester.scm`. However, it won't be able to compare against the expected results until you generate `.expect` files. You can do so using the same process described above for mutation testing:

```
$ plt-r5rs tokenize-tester.scm < test-file.scm > test-file.tokens.expect
$ plt-r5rs parse-tester.scm < test-file.scm > test-file.datums.expect
```

Make sure to manually check the contents of the resulting files to ensure they contain what you expect.

9 Submission

All code used by your lexer must be located in `lexer.scm` or the provided `distribution.scm`, which you may not modify.

All code used by your parser must be in `parser.scm` or the provided `distribution.scm`, except that your parser must use `read-token` from `lexer.scm`.

We will test your lexer and parser individually as well as together. Thus, your code must adhere to the API and conventions described in this spec. You may not assume that any files other than the provided `distribution.scm` are present when your lexer is tested. You may not assume that any files other than the provided `distribution.scm` and either your own `lexer.scm` or our solution are present when your parser is tested.

Submit `lexer.scm`, `parser.scm`, and your own test files to the autograder before the deadline. You may submit up to 30 files for the pattern `test*.scm` (do not submit `*.expect` files). If you have more test files than the limit, you will need to choose a subset to submit to the autograder.

10 Frequently Asked Questions

- **Q: Do our error messages need to match those in the spec or instructor solution?**

  A: You do not have to match the error messages; you just need to invoke the `error` procedure. The Makefile and autograder ignore the actual content of the error messages and just check that an error was signaled at the appropriate time.

- **Q: When I run the test cases, I get an error like** `Expected | line ["\n|\n"] after token, got: "|\nn"`**. What does this mean?**

  This means that you are reading the wrong number of characters from the input stream. Some of the testing frameworks use the pipe `|` separator in the input to determine whether you are consuming the correct amount of input. To fix this, you'll likely need to replace a `read-char` with `peek-char` (or vice versa if you're not reading enough input).

- **Q: I am failing the "Public read-datum Tests with Solution lexer.scm and Different Token ADT Implementation" test case, but I am only using** `read-token` **from** `lexer.scm`**. What could be the problem?**

  A: The most common culprits we've seen are:

  1. Violating the interface of the token ADT. This test intentionally uses a different implementation of the ADT than the one provided, but with the same interface.
  2. Using the wrong equivalence or membership predicate. There are several areas where the behavior of `eq?` and/or `eqv?` is unspecified, and others where the results depend on the implementation details of the lexer and parser. See the discussion in the R5RS spec on `eq?` vs. `eqv?` vs. `equal?`, which is what `memq`, `memv`, and `member` use, respectively.

- **Q: When I try to construct a list that corresponds to an abbreviation symbol, it gets printed by DrRacket not as a list but in the abbreviated form. Is there a way around this?**

  A: You don't have to worry about this. This is just DrRacket's convention for printing out a list corresponding to an abbreviation, and it doesn't affect the test cases. You can use `car` and `cdr` to double check that what you're constructing has the right form:

  ```
  > (list 'quote 'a)
  'a
  > (car (list 'quote 'a))
  quote
  ```

(continues on next page)

(continued from previous page)

```
> (cdr (list 'quote 'a))
(a)
```

More generally, the Scheme datum represented by `(quote a)` has multiple valid external representations – `(quote a)` and `'a`. The interpreter chooses one valid representation to print, and it may not be the same as what the user typed in. Here are a few other cases of this:

```
> 3.
3.0
> 3.00
3.0
> '(1 2 . (3 4))
(1 2 3 4)
```