Project 2: Scheme Lexer and Parser

Due Fri Oct 6 at 8pm

Contents

Background **2**
- Lexer . . . 2
  - Lexical Analysis . . . 2
- Parser . . . 3
  - Recursive Descent Parsing . . . 3
- Scheme I/O Procedures . . . 4
- Distribution Code . . . 4
  - Source Files . . . 4
  - Example Files . . . 5
  - Test Harnesses . . . 5
  - Test Files . . . 5
  - Errors . . . 6
  - Token Representation . . . 6

Phase 1: Lexing Individual Token Types **6**
- Strings . . . 6
- Booleans . . . 7
- Characters . . . 7
- Numbers . . . 7
- Identifiers . . . 7
- Punctuators . . . 7
- Errors . . . 7

Phase II: Full Lexer **8**

Phase III: Parser **8**
- Simple Expressions . . . 8
- Compound Expressions . . . 8
- Errors . . . 9
- Testing . . . 9

Rules and Regulations **9**

Grading **9**

Submission **10**

In this project, you will implement a lexer and parser for the R5RS Scheme programming language. The main purpose of this exercise is to get practice reasoning about the lexical and syntactic structure of a language. Secondary goals are to gain experience writing Scheme code and to learn how to express a program in the functional paradigm.

This project must be written in R5RS-compliant Scheme. The officially supported interpreter for this project is Racket. 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[1] 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.

Background

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

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 ⟨syntactic keyword⟩, ⟨expression keyword⟩, and ⟨variable⟩ do not appear in our lexical specification. Instead, all three categories are lexed as ⟨identifier⟩.
- 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 ⟨number⟩ 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 ⟨delimiter⟩, but not necessarily by anything else.

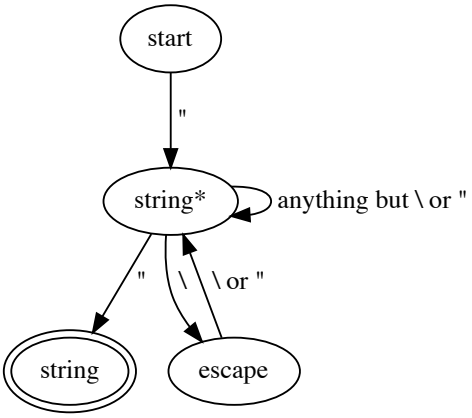
We will enforce this rigidly. Thus, inputs such as `asdf,` (including the comma) and `3a` are erroroneous, rather than being lexed as identifiers as in some Scheme implementations.

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, the following is an FSM for lexing strings in Scheme:

[1] Instructions: Windows; MacOS (e.g. `export PATH="/Applications/Racket v6.10/bin:$PATH"` for a Racket 6.10 installation on MacOS)



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 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.

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 Section 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:

$$S \rightarrow a\ S\ b \mid C$$
$$C \rightarrow c\ C \mid \varepsilon$$

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. 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.

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 and process input:

- `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
- `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

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.**

Distribution Code

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

`test-util.scm`	Utility functions used in test code.
`test-token-types.scm`	Testing framework for lexing individual token types. Read the documentation at the top of the file to determine how to use the framework.
`test-tokenize.scm`	Testing framework for lexing an entire Scheme file.
`test-read-simple.scm`	Testing framework for lexing and parsing simple Scheme expressions.
`test-read-compound.scm`	Testing framework for lexing and parsing compound Scheme expressions.
`test-read-datum.scm`	Testing framework for lexing and parsing all Scheme expressions.
`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 (`test-token-types.scm`, `test-tokenize.scm`, `test-read-simple.scm`, `test-read-compound.scm`, and `test-read-datum.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 test-token-types.scm < string-tokens.in > string-tokens.out
```

Test Files

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

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 may 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.

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.

Strings

Start by reading over the starter code for `read-string`. It checks to make sure that the initial character is a double quote, raising an error if 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 `test-token-types.scm` test driver. Some tests are contained in the file `string-tokens.in`, with the expected output in `string-tokens.expect`:

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

Booleans

Complete the `read-boolean-tail` procedure, which reads the tail end of a boolean literal and returns the token representation. Only the literals `#t` and `#f` are valid booleans. Raise an error if any other data is read.

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

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`.

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.

Identifiers

Read through the lexical specification for identifiers 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 the `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 part 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`.

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`.

Errors

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

Phase II: 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.

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 `test-tokenize.scm` calls this function and writes the result to standard output:

```
> plt-r5rs test-tokenize.scm < distribution.scm > distribution.scm.out
> diff distribution.scm.out distribution.scm.expect
```

Phase III: 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 the `tokenize` procedure.

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

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

Compound Expressions

Compound expressions consist of lists (including dotted lists), vectors, and abbreviations. Write the rest of the `read-compound-data` procedure, which reads and parses 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`.

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` or `read-datum-helper`.

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

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 `test-read-datum.scm`.

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 `test-read-simple.scm`, `test-read-compound.scm`, and `test-read-datum.scm` work on input files where every expression is repeated twice, using `read-datum` to read the first and `read` 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 should write your own test files, as the given test files only cover a small number of cases. We strongly suggest making use of the provided test harnesses and naming 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.

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` procedure, 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).

Any violations of these rules 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, 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.

Grading

The autograded portion of this project will constitute 90% of your total score, and the remaining 10% will be from hand grading. The latter will evaluate the comprehensiveness of your test cases as well as your programming practices, such as avoiding unnecessary repetition and respecting ADT interfaces. In order to be eligible for hand grading, your project must achieve at least half the points on the autograded portion (i.e. 45% of the project total).

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 a correct implementation of `lexer.scm` are present when your parser is tested.

Submit `lexer.scm`, `parser.scm`, and any of your own test files to the autograder before the deadline. **You must submit to the autograder -- pushing code to GitHub does not submit it to the autograder.** We suggest including a `README.txt` describing how to run your test cases.