Please read the following warning: **If you want to use any standard library procedures, ask first. You are free to use whatever is used in the sample code, or in this handout. If you have any questions, please, we don't mind if you ask. But if you use a library function without approval you will get a zero on this entire assignment.**

This problem set is intended to introduce you to recursion and list processing. We will tackle the problem of reading a scheme file–a structureless sequence of characters, and determining the expression–if any–represented by this character sequence. You will implement a toy version of the *read* part of the *read eval print loop.* This process usually has two distinct phases, called *lexing* and *parsing.*

1 Lexing

The first phase, *lexical analysis*, or *lexing*, breaks up a sequence of characters into chunks, called *tokens*, each with distinct meaning. Tokens are strings, numbers, symbols, parentheses, etc. Not only are we breaking the string up into chunks, but we are distinguishing different types of chunks from eachother. Consider the expression:

```
(string-ref "abcdefg" 1)
```

It contains the following tokens: a left paren, the symbol `string-ref`, the string `"abcdefg"`, the number `1`, and a right paren. Lexing is responsible for determining where one token ends and the the next begins (e.g. that the substring `1)` isn't a single token), what kind of token each substring represents (e.g. that `1` is a number, and not just an awful variable name), and its value (e.g. decode the string of characters `12345` into the number 12345, create a string whose contents is `abcdefg`, create a *symbol*–these are slightly different objects from strings–named `string-ref`.

1.1 Tokens

Your lexer will need to recognize the following types of tokens: strings, numbers, symbols, left parens, and right parens. There are a few more types of scheme tokens that you will need to know about, but that your lexer doeesn't have to recognize, such as booleans like `#t` and `#f` or characters like `#\a`.

1.1.1 Strings

You will take input in the form of a string, and also be expected to parse scheme code containing string literals. A string literal is, with one caveat, a bunch of characters with quotes on either side. The quotes are not part of the string. `"abcd"` denotes a string with 4 characters, whose first is `#\a` and whose last is `#\d`.

How do we write a string with a quotation mark in it? With our one caveat: escaping. `"a\"b"` is a string with three characters, the second of which is a double quotation mark. Similarly, inside a string literal, `\n` and `\t` denote newlines and spaces. And the way you write a string containing a backslash is with `\\`. **Your lexer does not need to handle escapes.**

Strings are basically arrays of characters, which means, like arrays, they support random access:

```
> (string-ref "abcdefg" 2)
#\c
```

Again, note the syntax for characters. Your lexer won't need to recognize this syntax, but you might need to write character literals in your program. Unlike Python, but like other languages, characters are **not** just single-element strings. Substrings can be extracted as follows:

```
> (substring "abcdefg" 0 2)
"ab"
> (substring "abcdefg" 1 1)
""
```

As with all programming, make sure to be careful about off-by-one errors.

1.2 Symbols

At some level of oversimplification, *symbols* are what the scheme evaluator interprets as variable names. They look like strings without quotation marks. Consider:

```
> (define a 1)
> a
1
> "a"
"a"
> (eq? (string->symbol "a") 'a)
#t
> (eval "a")
"a"
> (eval (string->symbol "a"))
1
```

When you type a symbol into the evaluator, it looks up the corresponding variable. But when you type a string, it just returns that string. There is a function `string->symbol` to turn a string into an identical looking symbol. We will require the following simplified format for symbols: the first character must be a letter, and then all subsequent characters must be found in the string `symbol-characters` provided in the source file. Of course, a real scheme implementation accepts `+`, `-`, etc. as symbols, but we won't make you implement those special cases.

Also, the input `myvar` is a single symbol, `not` two symbols `my` and `var`, and not a sequence of five single-character symbols. *Once you start consuming a symbol, you must keep going as far as you can.*

1.2.1 Numbers

You are only required to handle nonnegative base 10 integers. That is, strings of consecutive decimal digits. You are also required to produce an actual numeric value. Do *not* use the `string->number` function from the standard library. If you absolutely want to, you can write your own version of it, but this isn't necessary. We convert strings to numbers by reading digits left to right, and we also recognize numbers by reading characters right to left. So why not keep an accumulator as an argument to your `consume-number` procedure while you scan through the string?

Also note that numbers follow a similar rule to symbols: you must read as far in the string as you can, once you start reading a number. `10` is a single number, not a 1 and a 0. Furthermore, numbers must be delimited: `12abcdef` is not a number, and is not a symbol, but furthermore, it is not a sequence of a number followed by a symbol. Numbers must be followed by *delimiter* characters–whitespace, parens, or quotes.

1.2.2 Parens

All the previous tokens have some corresponding value to represent them. From a string of characters denoting a valid symbol, we create a symbol. From a string of digits, we create a number. How should we represent the parentheses at tokens? We can't use a placeholder string, like `"left-paren"`, because then how could we handle input containing this string?

If you look at the definition of the variables `left-paren` and `right-paren` in `parser-defns.rkt`, you will see some weirdness. Their values are the result of calling a special function called `string->uninterned-symbol`. Uninterned symbols compare not equal to all other symbols, and therefore all other values.

```
 > left-paren
'|(|
```

These variables are how we represent the paren tokens.

2 Parsing

Of course, expressions are not just a formless sea of tokens. They have structure. Fortunately, our expressions are about as simple as they get: their only structure is nesting, and nesting is indicated by parentheses.

Recall, every expression is either an *atom*–a symbol, number, or string, or a *list* of expressions, which we write by putting parentheses around a sequence of expressions. Note the parentheses aren't part of a list, they are just how we write it down in text.

In a well-formed expression, every left paren is closed by some right paren, and every right paren closes some left paren, and these are the *only* requirements. So conversely, the only things that can go wrong are:

- We forgot to close a paren.
- We put a closing paren when there was nothing to close.

More than knowing *if* a right paren closes some left paren, we also need to know *which* left paren it closes, if we are to determine the structure of the expression. Without nesting, each right paren closes the most recent left paren. With nesting, each right paren closes the most recent *unclosed* left paren. So, we keep track of left parens in the order we find them, and forget them in reverse order. So naturally, we need a stack.

Our algorithm requires two things:

- A list of input tokens
- And a stack, which holds both tokens and intermediate subexpressions. Note that lists make very nice stacks. You can examine the top of the stack with `car`, pop an element with `cdr`, and push with `cons`.

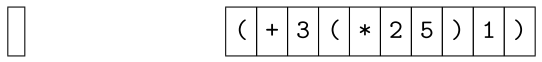
The stack holds both open parentheses and subexpressions. Gradually these subexpressions are joined into the full expression. This is done by the following algorithm:

- Look at the first token on the input list.
- Unless it's a closing paren, push it onto the the stack.
- If it IS a closing paren, pop items off the stack one by one, keeping them in a list, until we find an opening paren. The list of items we've been keeping is the list that was started by the opening paren, so replace it with the list on the stack. This should probably be a helper procedure.
- In both cases, advance to the next input token.

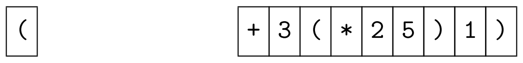
At the end, you should be left with one item on the stack, your parsed expression. You'll have to figure out how and where to detect and signal errors.

2.1 Example

In the following diagrams, the stack is shown on the left, with the top of the stack being furthest to the right. The input token list is shown on the right, with its first element to the left. We start with an empty stack, and the full list of tokens:



Then we read the first token and push it on the stack:



And another one:

$\boxed{(}\boxed{+}$ $\qquad\qquad$ $\boxed{3}\boxed{(}\boxed{*}\boxed{2}\boxed{5}\boxed{)}\boxed{1}\boxed{)}$

...And we keep doing this until we encounter a closing paren.

$\boxed{(}\boxed{+}\boxed{3}\boxed{(}\boxed{*}\boxed{2}\boxed{5}$ $\qquad\qquad$ $\boxed{)}\boxed{1}\boxed{)}$

Now the fun part. We read a closing paren, so we pop everything off the stack until the most recent opening paren, and collect it into a list:

$\boxed{(}\boxed{+}\boxed{3}\boxed{\texttt{(* 2 5)}}$ $\qquad\qquad$ $\boxed{1}\boxed{)}$

Note that when a paren appears alone in a box, it means a paren token, and when a parenthesized expression appears in a single box, it means that stack slot holds a list instead of a token. We have one more regular token to read:

$\boxed{(}\boxed{+}\boxed{3}\boxed{\texttt{(* 2 5)}}\boxed{1}$ $\qquad\qquad$ $\boxed{)}$

And then a final closing paren, to do the final collection:

$\boxed{\texttt{(+ 3 (* 2 5) 1)}}$ $\qquad\qquad$ $\boxed{}$

3 Assignment

You are provided `parser.rkt`, which contains the basic skeleton of a tokenizer, in the function `lex`. It knows to distinguish whitespace from numbers and parens, but nothing more, and instead of ignoring whitespace and converting numbers, it just returns a list of chunks. It also contains an empty function `parse`. It depends on `parser-defns.rkt`, also provided, which contains definitions of useful constants, helper functions, and defines two exception types that you will need to raise on invalid input.

Your assignment is to implement `lex` and `parse`. The file you submit must be called `parse.rkt` and must export the same names as the provided sample code, but of course you don't *have* to start with our code. Your code should pass all the tests in `parser-test.rkt`. You may modify `parser-defns.rkt` and `parser-test.rkt`, but we will run your code with unmodified versions of these files.

You are may use **only** the standard library functions mentioned in this handout (unless otherwise noted) and used in the sample code. For anything else, please check with us first. **If you use any other standard library functions without asking, you will get a zero on the entire assignment.**

3.1 Lexer

A good way to implement the lexer is by adding cases to the `dispatch` function provided. Note That you must also change the existing ones, since their behavior is incorrect.

`lex` must take an input string and return a list of tokens. The last token of the list must always be `end-of-input` (which will save you some grief when you write the parser). Your tokenizer must:

- Recognize symbols, and create them using `string->symbol`
- Recognize strings, and correctly handle the escape sequences `\\`, `\n`, `\t`, and `\"`. For this it might be wise to implement a separate `unescape` procedure.
- Recognize numbers, and convert them **without** using `string->number`
- Reject invalid input by raising the `lex-error` exception.

There is one more point that requires clarification: symbols and numbers must be followed by whitespace, a paren, or a quotation mark if they're not at the end of the input. Thus, input like `1abc` is not allowed. It's neither a symbol nor a number, and this extra rule is to keep it from being interpreted as two tokens.

3.2 Parser

Implement `parse`, which takes a token list and returns a nested list structure. Note that all tokens, besides parens, are the same to the parser. It doesn't care. If provided invalid input, `parse` should raise the `parse-error` exception.