Top-Down Parser Scheme

For this homework assignment, we will be extending the example from the previous programming assignment (the lexer in ruby), and we will be implementing a top-down parser in scheme. We want to use the technique that we reviewed in class (one "function" or procedure for each rule in our grammar). I have slightly rewritten the rules of the grammar so that it might be easier for us to write this program, however, the meaning has not changed. Find the rules below:

<pgm>	-->	<stmt>+
<stmt>	-->	<assign> | print <exp>
<assign>	-->	<id> = <exp>
<exp>	-->	<term> <etail>
<etail>	-->	+ <term> <etail> | - <term> <etail> | EPSILON
<term>	-->	<factor> <ttail>
<ttail>	-->	* <factor> <ttail> | / <factor> <ttail> | EPSILON
<factor>	-->	(<exp>) | <int> | <id>
<int>	-->	{0...9}+
<id>	-->	{a...zA...Z}+
<whitespace>	-->	(any whitespace)

I've uploaded a skeleton to get you started. Below are a few helper functions that you should use, with brief explanations of each.

For those that did the extra credit for the previous homework assignment, you should have a head start here. I'm simply reading in the tokens generated from a lexer and stored in a file, and creating a list of token/lexeme pairs. The name of this list (global variable) is "tokens". The path will be different on your machine.

```
(define tokens (file->lines
(string->path"F:\\Dropbox\\auburn_faculty\\RacketPrograms\\tokens")));;home computer
```

The following Procedure simply gets whatever the current token is (think about the book example, this is a similar concept) as a list with two items in it. In my example, the token name is the first list item and the lexeme is the second list item (just because that's the way I saved the tokens to my file. If you saved them in reverse order, you will need to keep that in mind when performing your checks (see ID and INT below).

```
(define current_token
  (lambda ()
    (regexp-split #px" " (car tokens))
    );end lambda
  );end current_token
```

Top-Down Parser Scheme

The following procedure removes the token at the beginning of the list so that current token always looks at the correct token. They are meant to be used together. For example, advance to the next token (next_token procedure), not let me see the next token (current_token procedure). In addition to advancing to the next token, this procedure will also “ignore” whitespace. If it encounters a whitespace token, it simply advances again so that current_token will never retrieve a whitespace token. If this procedure ever encounters an empty token list (no eof) it will exit with an error message.

```
(define next_token
  (lambda ()
    (set! tokens (cdr tokens))
    (cond
      ((null? tokens)
       (display "No more tokens to parse!")
       (newline)
       (display "Exiting prematurely, no eof found")
       (newline)
       (exit));end null tokens
      (else
       (cond
         ((equal? (car (current_token)) "whitespace")
          (next_token);call yourself again
          )
         );end cond
       );end else
      );end cond
    );end lambda
  );end next_token
```

I’ve also created procedures for the int (numeric) and id (identifier) rules. They are listed below. You are responsible for creating any additional necessary functions for the parser to correctly parse the grammar above (It’s the same grammar from the previous assignment). Notice that if either of the procedures is called and do not encounter the correct type of token, they display an error message, but otherwise continue parsing. All of your procedures should do the same.

```
(define int
  (lambda ()
    (display "Entering <int>")
    (newline)
    (cond
      ((equal? (car (current_token)) "int") ;← my first list item is the token name
       (display "Found ")
       (display (second (current_token))) ; ← my second list item is the lexeme
```

```
        (newline)
        (next_token)
        (display "Leaving <int>")
        (newline)
        );end first equality check in condition
       (else
        (display "Not an int token: Error")
        (newline)
        );end else statement
       );end condition block
     );end lambda
   );end int


 (define id
   (lambda ()
     (display "Entering <id>")
     (newline)
     (cond
       ((equal? (car (current_token)) "id") ;← my first list item is the token name
        (display "Found ")
        (display (second (current_token))) ; ← my second list item is the lexeme
        (newline)
        (next_token)
        (display "Leaving <id>")
        (newline)
        );end first equality check in condition
       (else
        (display "Not an id token: Error")
        (newline)
        );end else statement
       );end condition block
     );end lambda
   );end id
```

Finally, I've placed a few procedure stubs in the file for you (scheme_parser.rkt) with comments to help you think about how to proceed.

What's left for you

1) Write procedures that can correctly interpret all of the rules

2) Your program should have a way to “gracefully” handle errors so that it at least finishes parsing and displays some message on errors.
3) You need to provide a case for eof in the first rule. The program should exit once it encounters this in the pgm rule.

Error Handling

You need to think about how to handle errors. If you simply print that an error was found, will the program ever terminate?

Bonus – 10 points

Track how many parsing “errors” were encountered and print the number of errors when your parser terminates. If you encounter 0 errors, display a message that the parse was successful. If you encounter errors, display a message that the parse completed, but with errors, and indicate how many.

Helpful Tips

It might be helpful to use Dr. Racket to complete this assignment. It is a gui/interpreter for scheme. You can download it here: https://racket-lang.org

There are versions for every operating system.

Here are some additional resources that might be helpful. The reference is more complete, but the guide might be easier to swallow at first.:

https://docs.racket-lang.org/guide/index.html

https://docs.racket-lang.org/reference/index.html

I would at least have a look at the following sections. You should find the “cond” and the “lambda” constructs particularly useful for this assignment.:

https://docs.racket-lang.org/reference/if.html

https://docs.racket-lang.org/reference/lambda.html

My suggestion is to use Dr. Racket, and each time you finish one procedure, test it in interactive mode to ensure that it is working correctly before you move on. You may want to create stubs for other required procedures that simply display messages so that you at least know you are calling procedures at appropriate times.

We’ll go over Scheme and Dr. Racket in class together.