CSE 341: Homework 4

due 2/12/01

All functions you are asked to implement and turn in should be typed and tested with Miranda. With each function turn in the source code listing as well as sample output for a demonstrative set of tests. Do not forget to test *end cases*. Use the `turnin` program to turn in an electronic copy of your source code.

Part I: (Not to be turned in.)

Try this question out before section.

1. Write function `decode_hex_string` (like Homework 1) that takes a string and converts it as follows:

   - "+" gets converted to whitespace.
   - "%XX", where `XX` is a two digit hex number, gets converted to the character that it represents.
   - All other characters are unchanged.

   For example:

   ```
   decode_hex_string "In+hex+5+%2b+5+is+%27a%27."
         => "In hex 5 + 5 is 'a'."
   ```

   Use pattern matching.

Part II: (Not to be turned in.)

These are some sample problems that we will be discussing in section.

Given the following type definition:

```
paren ::= LP num | RP num
```

Here, `(LP 1)` represents a left parenthesis of type 1 and `(RP 2)` represents a right parenthesis of type 2. We will be looking at checking to see if lists of these parentheses are balanced. For example `[LP 1, LP 2, RP 2, RP 1]` is balanced and corresponds to a string like `"([])"`.

1. Write a function, `tokenize_parens`, which takes a string of `parens` (which might or might not be balanced, and turns it into a list of our `paren` type.

   ```
   tokenize_parens :: [char]->[paren]
   ```

   You should support the following three types of parentheses: `()`, `[]`, and `/**/`. For example:

   ```
   tokenize_parens "()[/*()[]*/]"
         => [LP 1,RP 1,LP 2,LP 3,LP 1,RP 1,LP 2,RP 2,RP 3,RP 2]
   ```

If any non-parentheses are in the string, give an error.

2. Write a function, `balanced`, which takes a list of `paren` and returns `True` if it is balanced and `False` if it is not. For example:

   ```
   balanced (tokenize_parens "()[/*()[]*/]")
          => True
   ```

Part III: (Due Monday, 2/12/01. Turn this in.)

Complete the following. If you encounter an error in the input to any of your functions, you should use the built in `error` function to announce what type of error occurred. **You must turn in output of your code on a demonstrative set of test cases.**

The first part of implementing a Scheme interpreter is parsing the Scheme code and turning it into some intermediate representation. In Scheme, this intermediate representation is just an S-expression which is either a value, a symbol, or a list of S-expressions. This facility is implemented in the primitive `read` function in Scheme.

For this assignment we are going to be concentrating on the reading and parsing of the scheme expressions. In this assignment we will implement the `scheme_read` function in Miranda to read and parse scheme instructions and return a tree structure of the data much as `read` does in Scheme.

You are to support the following types:

numbers For example `345` (You may restrict your attention to integers).

symbols For example `hello`.

strings For example `"hello"`. You do not have to support C-style escaping, `"\n\t\b\\"`.

lists List should be car/cdr lists with car elements of any of these four types. For example `(a b (10 11) ("hello"))`.

You should also support the quote symbol `'`. Note that all the quote symbol is is a transformation from `(a b '(c d))` to `(a b (quote (c d)))`. Quoting symbols is the same, `(10 'a 'b)` transforms to `(10 (quote a) (quote b))`.

This specification is deliberately a little bit vague. The reason for this is because part of your task with completing this assignment is to design suitable data types yourself. If you have questions about what you should do with a certain input, try it in Scheme (see what Guile does with it). To do this you may wish to use the Scheme `read` function.

1. (10 points) Implement the lexical analyzer. Write the function

   ```
   tokens :: [char]->[token]
   ```

   The function `tokens` takes a string as input and returns a list of tokens as output. You will need do define the type of `token`. The idea is that you want to turn the character string into a list of data that is easer to manipulate. This is the process of lexical analysis. For example, if we defined `token` as

```
token ::= RightParen | LeftParen | StringToken [char] | NumToken num | ...
```

then we would expect

```
tokens "(25 3.1)"     => [LeftParen, NumToken 25, NumToken 3.1, RightParen]
```

You need to complete the definition of the `token` type and implement `tokens`.

2. (15 points) The next step is writing the parser which will take a list of tokens and return a list of scheme expressions. You goal is to write the function

   ```
   parse :: [token]->[expr]
   ```

   where `expr` is a user defined type that will hold a single scheme expression. As stated above, the expressions read it should either be strings, numbers, symbols, or lists of expressions. Do not forget to implement `'` by transforming to `quote`.

   Hint 1 You might want to define the type `expr` like

   ```
   expr ::= StringExpr [char] | NumberExpr num | ...
   ```

   Hint 2 There are many ways to write a parser. A *shift-reduce* parser is a stack based parser that at each step will either *reduce* tokens and expressions already on the stack or it will shift the first unread token off the list of yet unparsed tokens onto the top of the stack. When you run out of tokens to shift and you cannot make any more reductions you are done. You can return your stack as the parsed answer. Note that if there is a parse error (i.e. bad input, for example `"(1)("` where the extra parentheses is a parse error) then you will be left with un-reduced tokens on the stack in addition to expressions.

   It is recommended that you use a shift reduce parser in your implementation of `parse`. To do so you will want to define a new data type that can be either a `token` or an `expr` as:

   ```
   tokexpr ::= Tok token | Expr expr
   ```

   Then you will want to define the *shift-reduce* parser as

   ```
   parse_shift_reduce :: [tokexpr]->[token]->[tokexpr]
   ```

   This is a function that takes as arguments the stack and the list of unused tokens (in that order) and returns the final stack. It remains for you to call `parse_shift_reduce` in `parse` and massage the output into the desired form.

3. (5 points) Now implement `scheme_read` to take a string that contains one Scheme expression and return a `expr` for that expression. If there are more than one expression in the string then read should announce the error with the `error` function. The type of your function should be

   ```
   scheme_read :: [char]->expr
   ```

   For example,

   ```
   scheme_read "1 2 3"     => error, too many expressions. read
   scheme_read "3"         => NumberExpr 3
   ```

```
scheme_read "(2 (3)"     => error, parse error.
scheme_read "(2 (3))"    => not an error, ??
```

The last example is not an error since there is only on expression (however it is nested). Your actual result will vary from implementation to implementation; however, what it represents is the list with the first element as the number 2 and the second element as a list containing only 3.