CSE 341, Autumn 2010
Assignment #7 - Scheme: BASIC Expression Parser
Due: Friday, November 19, 2010, 9:30 AM

In this assignment you will write a set of procedures for parsing various token sequences that appear in programs written in the BASIC language. The code that we write in this assignment will be used as part of a larger project in our next assignment, Homework 8, which will be an interpreter for a subset of BASIC syntax (including but not limited to expressions). Include your answers in a file named `parser.scm`. A skeleton of this file is provided on the web site.

Background Information about Parsing:

A ***grammar*** is a set of syntax rules that collectively describe a language. ***Parsing*** (*syntactic analysis*) is the process of analyzing a text, made of a sequence of tokens, to determine its grammatical structure with respect to a given grammar. A parser is a component in an interpreter or compiler that checks for correct syntax. In a compiler, the parser often builds a tree, list, or other hierarchical structure from the input tokens. An interpreter instead runs the code as it parses it. Parsing is often considered to be distinct from *lexical analysis*, which is the process of reading each character of text in a program and splitting up those characters into a set of *tokens*.

A *top-down* parser looks at the sequence of tokens in order from start to finish, trying to "expand" its understanding of the program elements from large items (such as a program or function) down to small ones (such as a single expression). By contrast, a *bottom-up* parser begins by identifying the smallest and simplest symbols in the program first, then expanding outward from them to discover their context within larger forms such as expressions, statements, functions, etc.

In this assignment you will write a particular subcategory of top-down parser known as a *recursive descent parser*. In a recursive descent parser, you begin with a grammar for the language of interest, and you use that grammar to write a series of procedures, one for each major non-terminal symbol and/or production rule in that grammar. As your parser reads tokens of input, you use the sequences and values of those tokens to decide which of your procedures to call.

Since some forms can contain themselves (for example, an expression can contain other expressions), you sometimes end up with one of your production procedures calling itself; or you end up with a chain of calls such as $a \rightarrow b \rightarrow c \rightarrow a$. In this sense the parser is *recursive*. Its *descent* is the path that it walks through the grammar rules and their associated functions as it parses the various tokens of the program that you feed to it.

Our Language's Grammar:

Our grammar below is in a format called *Extended Backus-Naur Form* (EBNF). It has the following production rules:

<*test*> ::= <*expression*> ("<" | ">" | "<=" | ">=" | "=" | "<>") <*expression*>
<*expression*> ::= <*term*> {("+" | "-") <*term*>}
<*term*> ::= <*element*> {("*" | "/") <*element*>}
<*element*> ::= <*factor*> {"^" <*factor*>}
<*factor*> ::= <*number*> | ("+" | "-") <*factor*> | "(" <*expression*> ")" | <*f*> "(" <*expression*> ")"
<*f*> ::= SIN | COS | TAN | ATN | EXP | ABS | LOG | SQR | RND | INT

Here are some details about EBNF:

- The pipe character ("|") means "or." The final rule says that an "<f>" is either SIN or COS or TAN, and so on.
- Curly braces indicate "0 or more of." For example, the production for <element> indicates that it is composed of a <factor> followed by 0 or more occurrences of "^ <factor>". So it would match sequences like this:

  <factor> ^ <factor> ^ <factor> ^ <factor> ^ <factor>
- Tokens shown in quotes in the grammar represent literal values. In Scheme these will be stored as symbols.
- Parentheses are used to group subparts of a rule. For example, the <expression> rule uses parentheses to indicate that terms are separated by either a plus or a minus, as in:

  <term> + <term> + <term> - <term> + <term> - <term> - <term>

You should define a procedure to parse each of the five major elements described above. The sixth rule is slightly different (described below). Each of your procedures will take a list of tokens as its parameter. It should consume tokens from the list and replace those tokens with the value obtained by evaluating the tokens. For example, `parse-term` evaluates the multiplicative operators `*` and `/`. If it is passed the list `(2.5 * 3.4 / 2 THEN 210)`, it will evaluate `2.5 * 3.4 / 2`, obtain `4.25` and replace the tokens it parsed with `4.25`, returning the list `(4.25 THEN 210)`.

These procedures are "***greedy***" in that they will consume as many tokens from the front of the list as they can that can be part of the given grammar element. Above, the token `THEN` can't be part of a *term*, so the procedure stops consuming tokens when it encounters it. Though it is greedy, it should pay attention only to tokens that appear at the front of the list.

Parentheses are difficult to handle as symbols, so we'll use tokens `lparen` and `rparen` for left and right parentheses.

There are several cases where Scheme returns rational numbers when we would prefer to obtain real numbers. For example, division can produce such results as can exponentiation with integers where the exponent is negative:

```
> (/ 3 4)
3/4
> (expt 3 -2)
1/9
```

For these cases, you should call the `exact->inexact` procedure to convert the rational to a real number:

```
> (exact->inexact (/ 3 4))
0.75
> (exact->inexact (expt 3 -2))
0.1111111111111111
```

Procedures to Implement:

You are to write the following parsing procedures *(also see the end of this spec for a suggested development strategy)*:

1. `parse-factor`

Write a procedure `parse-factor` that parses a factor at the front of a list, replacing the tokens that were part of the factor with the numeric value of the factor. Recall the grammar rule for a factor:

<*factor*> ::= <*number*> | ("+" | "-") <*factor*> | "(" <*expression*> ")" | <*f*> "(" <*expression*> ")"

As indicated in the grammar, a factor is either a simple number or a sign followed by a factor (e.g., the negation of a number) or a parenthesized expression or a parenthesized call on a function. For example:

```
> (parse-factor '(3.5 2.9 OTHER STUFF))
(3.5 2.9 OTHER STUFF)
> (parse-factor '(- 7.9 3.4 * 7.2))
(-7.9 3.4 * 7.2)
> (parse-factor '(lparen 7.3 - 3.4 rparen + 3.4))
(3.9 + 3.4)
> (parse-factor '(SQR lparen 12 + 3 * 6 - 5 rparen))
(5)
> (parse-factor '(- lparen 2 + 2 rparen * 4.5))
(-4 * 4.5)
```

Keep in mind that your procedures should be greedy, but not overly greedy. For example:

```
> (parse-factor '(- 13 - 17 - 9))
(-13 - 17 - 9)
```

Only the first combination of minus and number is turned into a negative number. The method should process just one factor at the front of the list, not multiple factors.

2. `parse-element`

Write a procedure `parse-element` that parses an element at the front of a list, replacing the tokens that were part of the element with the numeric value of the element. Recall the grammar rule for an element:

<*element*> ::= <*factor*> {"^" <*factor*>}

As indicated in the grammar, an element is a series of one or more factors separated by the token `^` which indicates exponentiation. Your procedure should be "greedy"; it should consume as many tokens as possible that could be part of an element. The `^` operators should be evaluated left-to-right. You can call the standard `expt` procedure to compute the exponent. As noted above, the result can be a real number when the exponent is negative, so you need to call `exact->inexact` in that case to convert from a rational to a real number. Below are several examples of `parse-element`'s intended behavior:

```
> (parse-element '(2 ^ 2 ^ 3 THEN 450))
(64 THEN 450)
> (parse-element '(2 ^ 2 ^ -3 THEN 450))
(0.015625 THEN 450
> (parse-element '(2.3 ^ 4.5 * 7.3))
(42.43998894277659 * 7.3)
> (parse-element '(7.4 + 2.3))
(7.4 + 2.3)
> (parse-element '(2 ^ 3 ^ 4 ^ 5 2 ^ 3 4 ^ 5))
(1152921504606846976 2 ^ 3 4 ^ 5)
```

3. `parse-term`

Write a procedure `parse-term` that parses a term at the front of a list, replacing the tokens that were part of the term with the numeric value of the term. Recall the grammar rule for a term:

<*term*> ::= <*element*> {("*" | "/") <*element*>}

As indicated in the grammar, a term is a series of elements separated by the tokens `*` and `/` (multiplication and division). Your procedure should consume as many tokens as possible that could be part of a term. The `*` and `/` operators should be evaluated left-to-right. Scheme will potentially be returning a rational number when we use division, which we don't want. So for each division, return the value obtained by passing the result to the standard Scheme procedure `exact->inexact`, as in `(exact->inexact 7/8)`, which returns `0.875`. For example:

```
> (parse-term '(2.5 * 4 + 9.8))
(10.0 + 9.8)
> (parse-term '(38.7 / 2 / 3 THEN 210))
(6.45 THEN 210)
> (parse-term '(7.4 * lparen 2.4 - 3.8 rparen / 4 - 8.7))
(-2.59 - 8.7)
> (parse-term '(3 / 4 + 9.7))
(0.75 + 9.7)
> (parse-term '(2 * 3 * 4 + 3 * 8))
(24 + 3 * 8)
> (parse-term '(24 / 2 - 13.4))
(12.0 - 13.4)
```

4. `parse-expression`

Write a procedure `parse-expression` that parses an expression at the front of a list, replacing the tokens that were part of the expression with the numeric value of the expression. Recall the grammar rule for an expression:

<*expression*> ::= <*term*> {("+" | "-") <*term*>}

As indicated in the grammar, an expression is a series of terms separated by the tokens `+` and `-` (addition and subtraction). Your procedure should consume as many tokens as possible that could be part of an expression. The `+` and `-` operators should be evaluated left-to-right. For example:

```
> (parse-expression '(12.4 - 7.8 * 3.5 THEN 40))
(-14.9 THEN 40)
> (parse-expression '(2 + 3.4 - 7.9 <= 7.4))
(-2.5 <= 7.4)
> (parse-expression '(3 * 4 ^ 2 / 5 + SIN lparen 2 rparen))
(10.50929742682568)
> (parse-expression '(15 - 3 - 2 foo 2 + 2))
(10 foo 2 + 2)
```

5. parse-test

Write a procedure `parse-test` that parses a test at the front of the list, replacing the tokens that were part of the test with the boolean value of the test (`#t` or `#f`). Recall the grammar rule for a test:

<*test*> ::= <*expression*> ("<" | ">" | "<=" | ">=" | "=" | "<>") <*expression*>

As indicated in the grammar, a test is an expression followed by a relational operator, then an expression. For example:

```
> (parse-test '(3.4 < 7.8 THEN 19))
(#t THEN 19)
> (parse-test '(2.3 - 4.7 ^ 2.4 <> SQR lparen 8 - 4.2 ^ 2 * 9 rparen FOO))
(#t FOO)
> (parse-test '(2 ^ 4 = 4 ^ 2))
(#t)
```

The **sixth grammar rule** has a list of **function** names. Because it doesn't mention any other grammar elements, it doesn't need its own parsing procedure. You are welcome to define a helper procedure that processes it, but it's not a requirement. In evaluating expressions involving these functions, you will need to translate the named function into a Scheme procedure. The skeleton file includes the following **association list** that will help you to accomplish this task:

```
(define functions
  '((SIN . sin) (COS . cos) (TAN . tan) (ATN . atan) (EXP . exp) (ABS . abs)
    (LOG . log) (SQR . sqrt) (RND . rand) (INT . trunc)))
```

Each element of the association list is a dotted pair where the `car` is a key and the `cdr` is a value (like a `Map` in Java). Scheme has a standard procedure called `assoc` that will search for a particular key. For example, given the list above, the expression `(assoc 'ATN functions)` returns `(ATN . atan)`. To get the `atan` symbol from this, you'd take the `cdr` of the pair object. Because it's an improper dotted pair and not a list, you use `cdr` instead of `cadr`. If the key is not found, as in the call `(assoc 'FOO functions)`, you get `#f` (false) as a result.

Error-Handling:

In your parse routines, you should test for potential **errors** and should call the `error` procedure if an error is encountered. The call should indicate which parsing routine detected the error. Execute one of the following five commands:

```
(error "illegal test")
(error "illegal expression")
(error "illegal term")
(error "illegal element")
(error "illegal factor")
```

Most errors will be detected by `parse-factor` since it is the lowest level construct in the grammar. In grading, we won't be concerned about which of your parsing methods detects the error. All that we care about is that your code generates the error (versus, say, calling `car` on an empty list). Below are some examples of calls that should produce errors:

```
(parse-term '(2.5 * 4 *))          ; no number after second *
(parse-factor '(foo bar))          ; the symbol foo is not a legal factor
(parse-factor '(- foo))            ; a minus must be followed by a factor
(parse-term '(3 * foo))            ; no number after *
(parse-factor '(lparen rparen))    ; needs an expression between parens
(parse-factor '(lparen 2 + 3 4))   ; no right paren when we encounter 4
```

Suggested Development Strategy:

First, write a simple version of `parse-factor`. Have it handle a number. This is a pretty trivial case, because it just puts the number back at the front of the list. But then have it handle the rule where a factor can be a sign followed by a factor. Whenever you see a non-terminal on the right side of a grammar rule, you know that it should be handled by a call on the corresponding parsing procedure. So this is a case where `parse-factor` is going to call `parse-factor`. If you code this correctly, you should be able to handle cases like this:

```
> (parse-factor '(2 + 2))
(2 + 2)
> (parse-factor '(- 2 + 2))
(-2 + 2)
> (parse-factor '(- - 2 + 2))
(2 + 2)
> (parse-factor '(- - - - - 2 2))
(-2 2)
> (parse-factor '(+ - - + + - + - + 2 2))
(2 2)
```

Second, write `parse-element`. It has to handle the case where there is just a factor and the case where that is followed by zero or more of an up-arrow followed by factor, as in:

```
> (parse-element '(2 + 2))
(2 + 2)
> (parse-element '(- - - 2 + 2))
(-2 + 2)
> (parse-element '(2 ^ 3 - 4))
(8 - 4)
> (parse-element '(- - 2 ^ - - - 3 * 17))
(0.125 * 17)
> (parse-element '(2 ^ 3 ^ 2 / 14))
(64 / 14)
> (parse-element '(- - - 7 ^ - - 4 ^ - - - 2 < 74.5))
(1.7346652555743034e-007 < 74.5)
```

(Remember that the grouping is left-to-right.)

Third, write `parse-term` and `parse-expression`. You could initially have `parse-term` handle just multiplication and `parse-expression` handle just addition. That makes them look a lot like `parse-element`.

Fourth, go back to `parse-factor` and have it deal with parenthesized expressions. Then deal with function calls.

Fifth, write `parse-test`.

Sixth, finish whatever you left unfinished (e.g., if you had `parse-expression` do just addition, then have it handle subtraction as well), including testing for errors.

Recall that you can use the **DrScheme debugger** to step through the execution of your various procedures to find bugs. You can also insert calls to the `display` procedure to view printed output on the console while your code is running.

If you produce any test cases or **testing code** for this assignment, you are welcome to share it with your classmates using our course discussion forum. Any shared testing code may test for external behavior but may not test any aspects of the internal correctness of the student's Scheme source code itself. The instructor and TAs reserve the right to remove testing code that we feel is inappropriate for any reason.

Grading and Submission:

Submit your finished `parser.scm` file electronically using the link on the class web page.

For reference, our solution is roughly 150 lines (70 "substantive" lines) long according to the course Indenter page. You don't need to match this number or even come close to it; it is just a rough guideline.

Your program should not produce syntax or runtime **errors** when executed. You may lose points if you name your procedures or top-level values incorrectly, even if their behavior is correct.

Your code should work for both basic and advanced cases. Perform your own **testing**, and remember to test edge cases.

You should not use any of Scheme's language features that involve **mutation**, such as the `set!`, `set-car!`, or `mcons` procedures. You are not allowed to use mutable lists or vectors in solving this problem. You should not define any Scheme **macros**. Your code should compile and execute properly using the **"Pretty Big"** language level of PLT Scheme. Otherwise you may use any Scheme library constructs you like to help you solve a problem unless those constructs are explicitly forbidden by this spec or the problem description.

As always, if the solution to one procedure is useful in helping you solve a later procedure, you should call the earlier procedure from the later procedure. You can include any testing code you develop (although it should be labeled as such). You may define other **global variables or procedures** not described in this document, as long as they are non-trivial and are used by more than one top-level procedure specified in this document.

If you write inner helper procedures, choose a suitable set of **parameters** for each one. Don't pass unnecessary parameters or parameters that are unmodified duplicates of existing bound parameters from the outer procedure.

The lists of tokens passed to your various procedures often contain various **symbols**. You should process and handle these symbols without converting them into strings first. (Treating symbols as strings everywhere is bad Scheme Zen.)

You are expected to use good programming **style**, such as naming, indentation/spacing, and avoiding redundancy when possible. Avoid long lines of over 100 characters in length; if you have such a line, split it into multiple lines. Recall that DrScheme is able to auto-indent your entire program for you simply by selecting code and pressing the Tab key.

In general, favor the `cons` procedure to grow lists versus the `concat` procedure (though the `concat` procedure is sometimes necessary and is not forbidden entirely). Favor the use of higher-order procedures such as `map` over manually iterating over / recursively processing a list.

Place a descriptive **comment** header at the top of your program along with a comment header on each procedure, including a description of the meaning of its parameters, its behavior/result, and any preconditions it assumes. If you declare a non-trivial or non-obvious inner helper procedure, also briefly comment the purpose of that helper in a similar fashion. Since the procedures in this assignment are larger and more elaborate, you should put comments within the procedures explaining the complex code, such as which grammar rule a particular part of the code is processing, etc.

Efficiency on a fine level of detail is not crucial on this assignment. But code that is unnecessarily computationally inefficient (such as code that performs an algorithm that should be $O(n)$ in $O(n^2 \log n)$ time) might receive a deduction.

Redundancy, such as recomputing a value unnecessarily or unneeded recursive cases, should be avoided. This assignment involves a lot of similar but not identical code, such as checking whether a given expression matches a particular pattern, then breaking apart the list to process it based on that pattern. Intelligently utilize helper procedures to capture common code to avoid redundancy. Avoid repeating large common subexpressions as much as possible; for example, the pseudo-code *"if (test) then (1 + really long expression) else (2 + really long expression)"* is better written as *"((if test then 1 else 2) + really long expression)"*.