An Interactive Desk Calculator
Project P2 of
Common Lisp: An Interactive Approach

Stuart C. Shapiro
Department of Computer Science
State University of New York at Buffalo

January 25, 1996

The goal of this project is to implement an interactive desk calculator. The moral of the project is that if you don't like the syntax of COMMON LISP, you can change it. This lesson can be appreciated by seeing how we could evaluate the arithmetic expression

$$\frac{10^{5^2} \times 3}{4} - 5 \times 6 - 25$$

at the end of each of several chapters:

4:
```
> (- (- (/ (* (expt 10 (expt 5 2)) 3) 4) (* 5 6)) 25)
7499999999999999999999945
```

19:
```
> (compute '(10 ^ 5 ^ 2 * 3 / 4 - 5 * 6 - 25))
7499999999999999999999945
```

20:
```
> (compute 10 '^ 5 '^ 2 '* 3 '/ 4 '- 5 '* 6 '- 25)
7499999999999999999999945
```

23:
```
> (compute 10 ^ 5 ^ 2 * 3 / 4 - 5 * 6 - 25)
7499999999999999999999945
```

29:
```
> 10 ^ 5 ^ 2 * 3 / 4 - 5 * 6 - 2 =
7499999999999999999999968
```

Each version translates an expression using some initial syntax into an expression in Cambridge Prefix notation, and then evaluates the latter expression. The versions differ in having the initial syntax closer and closer to the standard syntax of arithmetic expressions. This syntax is:

expression	::=	*term* | *expression* `{+|-}` *term*
term	::=	*factor* | *term* `{*|/}` *factor*
factor	::=	*operand* | *operand* `^` *factor*

This grammar ignores several complications such as unary `+` and `-`, embedded expressions enclosed in parentheses, and embedded calls to COMMON LISP functions. These complications are gradually introduced as the exercises progress.

The key function that does the translation is

```
(defun prefix (expression)
  "Returns the arithmetic EXPRESSION
   translated into Cambridge Prefix notation."
  ...)
```

The design of `prefix` is based on the idea of a *recursive descent parser* (Notice how this directly follows the above grammar.):

To get an expression:

1. Get the first term.
2. If there's no more to the expression, return the first term.
3. Else, the first element of the remaining expression will be `+` or `-`; call that the operator, remove it, and get the first term of the remainder, calling it the second term.
4. Combine the first term, the operator, and the second term into Cambridge Prefix notation, and put it back on the front of the expression.
5. Get an expression, and return it.

To get a term:

1. Get the first factor.
2. If there's no more to the expression, return the first factor.
3. Else, if the first element of the remaining expression is `+` or `-`, return the first factor.
4. Else, the first element of the remaining expression will be `*` or `/`; call that the operator, remove it, and get the first factor of the remainder, calling it the second factor.
5. Combine the first factor, the operator, and the second factor into Cambridge Prefix notation, and put it back on the front of the expression.
6. Get a term, and return it.

To get a factor:

1. Get the first operand.
2. If there's no more to the expression, return the first operand.
3. Else, if the first element of the remaining expression is not `^`, return the first operand.
4. Else, the first element of the remaining expression will be `^`; call that the operator, remove it, and get the first factor of the remainder, calling it the second argument.

5. Combine the first operand, the operator, and the second argument into Cambridge Prefix notation, and return it.

The procedures to get an expression, a term, and a factor should return two things:

1. an expression, term, or factor;
2. the rest of the expression.

COMMON LISP has a way for functions to return multiple values, but that is an advanced topic, so until that topic is reached, the technique we will use is to `cons` the expression, term, or factor onto the front of the rest of the expression, where it will look like an initial argument. Thus the major functions we will write will have the following specifications:

```
(defun prefix (expr)
  "Returns the expression in Cambridge Prefix."
  ...)

(defun enclose-expression (expr)
  "EXPR is a cons representing an expression in infix.
   Its first term is prefixed and enclosed as its first member.
   Returns EXPR entirely prefixed and enclosed in a list."
  ...)

(defun enclose-term (expr)
  "EXPR is a cons representing an expression in infix.
   Its first factor is prefixed and enclosed as its first member.
   Returns EXPR
   with its first term prefixed and enclosed in a list."
  ...)

(defun enclose-factor (expr)
  "EXPR is a cons representing an expression in infix.
   Returns EXPR
   with the first factor prefixed and enclosed in a list."
  ...)

(defun combine-expr (op x e)
  "OP is an operator.  X is its first operand.
   E is a list that starts with OP's second operand, Y.
   Returns E with (OP X Y) replacing Y."
  ...)
```

Here is a schematic trace of the entire procedure:

Translate to prefix `(10 ^ 5 ^ 2 * 3 / 4 - 5 * 6 - 25)`

1. Enclose a factor of `(10 ^ 5 ^ 2 * 3 / 4 - 5 * 6 - 25)`
    - 1.1. The first operand is `10`
    - 1.2. The operator is `^`
    - 1.3. Enclose a factor of `(5 ^ 2 * 3 / 4 - 5 * 6 - 25)`
        - 1.3.1. The first operand is `5`
        - 1.3.2. The operator is `^`
        - 1.3.3. Enclose a factor of `(2 * 3 / 4 - 5 * 6 - 25)`
            - 1.3.3.1. The first operand is `2`
            - 1.3.3.2. Return `(2 * 3 / 4 - 5 * 6 - 25)`
        - 1.3.4. The second argument is `2`
        - 1.3.5. Return `((^ 5 2) * 3 / 4 - 5 * 6 - 25)`
    - 1.4. The second argument is `(^ 5 2)`
    - 1.5. Return `((^ 10 ( ^ 5 2)) * 3 / 4 - 5 * 6 - 25)`
2. Enclose a term of `((^ 10 ( ^ 5 2)) * 3 / 4 - 5 * 6 - 25)`
    - 2.1. The first factor is `(^ 10 ( ^ 5 2))`
    - 2.2. The operator is `*`
    - 2.3. Enclose a factor of `(3 / 4 - 5 * 6 - 25)`
        - 2.3.1. The first operand is 3.
        - 2.3.2. Return `(3 / 4 - 5 * 6 - 25)`
    - 2.4. The second factor is `3`
    - 2.5. Enclose a term of `((* (^ 10 ( ^ 5 2)) 3) / 4 - 5 * 6 - 25)`
        - 2.5.1. The first factor is `(* (^ 10 ( ^ 5 2)) 3)`
        - 2.5.2. The operator is `/`
        - 2.5.3. Enclose a factor of `(4 - 5 * 6 - 25)`
            - 2.5.3.1. The first operand is `4`
            - 2.5.3.2. Return `(4 - 5 * 6 - 25)`
        - 2.5.4. The second factor is `4`
        - 2.5.5. Enclose a term of `((/ (* (^ 10 ( ^ 5 2)) 3) 4) - 5 * 6 - 25)`
            - 2.5.5.1. The first factor is `(/ (* (^ 10 ( ^ 5 2)) 3) 4)`
            - 2.5.5.2. Return `((/ (* (^ 10 ( ^ 5 2)) 3) 4) - 5 * 6 - 25)`
        - 2.5.6. Return `((/ (* (^ 10 ( ^ 5 2)) 3) 4) - 5 * 6 - 25)`
    - 2.6. Return `((/ (* (^ 10 ( ^ 5 2)) 3) 4) - 5 * 6 - 25)`

3. Enclose an expression of `((/ (* (^ 10 ( ^ 5 2)) 3) 4) - 5 * 6 - 25)`
    3.1. The first term is `(/ (* (^ 10 ( ^ 5 2)) 3) 4)`
    3.2. The operator is `-`
    3.3. Enclose a factor of `(5 * 6 - 25)`
        3.3.1. The first operand is `5`
        3.3.2. Return `(5 * 6 - 25)`
    3.4. Enclose a term of `(5 * 6 - 25)`
        3.4.1. The first factor is `5`
        3.4.2. The operator is `*`
        3.4.3. Enclose a factor of `(6 - 25)`
            3.4.3.1. The first operand is `6`
            3.4.3.2. Return `(6 - 25)`
        3.4.4. The second factor is `6`
        3.4.5. Enclose a term of `((* 5 6) - 25)`
            3.4.5.1. The first factor is `(* 5 6)`
            3.4.5.2. Return `((* 5 6) - 25)`
        3.4.6. Return `((* 5 6) - 25)`
    3.5. The second term is `(* 5 6)`
    3.6. Enclose an expression of `((- (/ (* (^ 10 ( ^ 5 2)) 3) 4) (* 5 6)) - 25)`
        3.6.1. The first term is `(- (/ (* (^ 10 ( ^ 5 2)) 3) 4) (* 5 6))`
        3.6.2. The operator is `-`
        3.6.3. Enclose a factor of `(25)`
            3.6.3.1. The first operand is `(25)`
            3.6.3.2. Return `(25)`
        3.6.4. Enclose a term of`(25)`
            3.6.4.1. The first factor is `(25)`
            3.6.4.1. Return `(25)`
        3.6.5. Enclose an expression of `((- (- (/ (* (^ 10 ( ^ 5 2)) 3) 4) (* 5 6)) 25))`
            3.6.5.1. The first term is `(- (- (/ (* (^ 10 ( ^ 5 2)) 3) 4) (* 5 6)) 25)`
            3.6.5.2. Return `((- (- (/ (* (^ 10 ( ^ 5 2)) 3) 4) (* 5 6)) 25))`
        3.6.6. Return `((- (- (/ (* (^ 10 ( ^ 5 2)) 3) 4) (* 5 6)) 25))`
    3.7. Return `((- (- (/ (* (^ 10 ( ^ 5 2)) 3) 4) (* 5 6)) 25))`
4. Return `(- (- (/ (* (^ 10 ( ^ 5 2)) 3) 4) (* 5 6)) 25)`

To simplify and to save space this has ignored the fact that the Cambridge Prefix version of `^` is `EXPT`.

Revised Sequence of P2 Exercises

You should do the p2 exercises in the order shown below, even if the numbering is now out of order.

12.5–6 Do as specified on p. 81.

12.3 Compile your `calculator` file by evaluating

`(compile-file "calculator")`.

12.4 Load your `calculator` file using the `load` function. Can you tell that you've loaded the compiled file instead of the source file?
Test `combine-expr` again.

17.33 Do as specified on p. 121.

17.36 Do as specified on p. 122.

17.35 Do as specified on pp. 121–122, but allow the exponentiation operator, `^`, to be in the expression also.

17.34 Write `enclose-expression` to meet the specifications shown above, assuming that the only operators are binary `+` and `-`, along with `*`, `/`, and `^`, and assuming that the input expression is a list whose only sublist may be the initial term.

18.28–29 Define `prefix` to take an expression that may have binary `+` and `-`, `*`, `/`, and `^`, and return the expression in Cambridge Prefix Notation.

19.10–13 Do as specified on p. 147.

19.14 Do as specified on p. 147, but note that, although `+` and `-` are names of Common Lisp functions, unary `+` or `-` may be the first element of a list, and that list should be modified.

19.15–16 Do as specified on p. 147.

20.3–5 Do as specified on p. 152.

21.9–10 Do as specified on p. 158.

23.6–8 Do as specified on p. 171.

24.8 Do as specified on p. 180.

29.33 Do as specified on p. 228.