Construct in Racket a parser for free-form Unicalc expressions. Use it to create a user-friendly CLI for Unicalc arithmetic. For features other than evaluation of expressions, you do only need to provide define, not let, let*, etc. The input grammar is shown below.

Each expression must be on a single line of input. Definitions are of the form

```
define $variable-name = expression
```

Otherwise the expression is a Unicalc expression to be evaluated. The parser should call the `ueval` evaluator that you constructed in earlier assignments produce the latter output on the command line. **Note that whitespace is implicitly allowed between major syntactic entities, but not within identifiers, or numerals.**

<table>
<thead>
<tr>
<th>Production</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>L → S</td>
<td>A line L is the start symbol of the grammar. A line L is either an expression S to be evaluated or a definition.</td>
</tr>
<tr>
<td>S → E { {&lt;space&gt;</td>
<td>/ } E }*</td>
</tr>
<tr>
<td>E → P</td>
<td>An elementary expression E is a primary expression P, optionally followed by a ^ (representing exponentiation) and an integer I.</td>
</tr>
<tr>
<td>P → C</td>
<td>A primary expression P is a coefficient C, a unit U, a variable V, or a parenthesized expression.</td>
</tr>
<tr>
<td>I → { +</td>
<td>-</td>
</tr>
<tr>
<td>D → 0</td>
<td>D represents a digit.</td>
</tr>
<tr>
<td>V → $ Letter {Letter</td>
<td>D}*</td>
</tr>
<tr>
<td>C</td>
<td>C represents a numeric coefficient, which can be any unsigned integer or floating numeral.</td>
</tr>
<tr>
<td>Letter → a</td>
<td>L represents a letter (which includes underscore _).</td>
</tr>
<tr>
<td>Letter U</td>
<td>A unit U is any sequence of letters.</td>
</tr>
</tbody>
</table>
Above, braces are used to group elements of the grammar. For example \{ + | - | ε\} means one alternative of a +, -, or nothing (the empty string). Likewise \{\ldots\}\* means 0 or more repetitions of the part inside the braces. For example the V production says that a variable is a $ followed by a Letter, followed by 0 or more Letters or Digits. This is meant to suggest use of *iteration* rather than recursion, although \{\ldots\}\* could be replaced with a new non-terminal X, and productions X \→ \ldots X | ε.

**Examples**, showing input line and both parser and evaluator output:

```plaintext
line: 1 foot/hour
parse: (/ (* 1.0 foot) hour)
normalized quantity: 8.46651371E-5 (meter)/(second)

line: 2 mph / (foot/sec)
parse: (/ (* 2.0 mph) (/ foot sec))
normalized quantity: 2.933333333333336

line: 1 (foot pound_force) / (newton meter)
parse: (/ (* 1.0 (* foot pound_force)) (* newton meter))
normalized quantity: 1.355793454465969

line: 3.14
parse: 3.14
normalized quantity: 3.14

line: 3.14^2
parse: (* 3.14 3.14)
normalized quantity: 9.8596

line: define $x = 10 acre
parse: (define $x (* 10.0 acre))
$x = 40467.10247438835 (meter meter)

line: $x
parse: $x
normalized quantity: 40467.10247438835 (meter meter)

line: $x / yard^2
parse: (/ $x (* yard yard))
normalized quantity: 48399.99999999999

line: 3.456 (joule/coulomb) / volt
parse: (/ (* 3.456 (/ joule coulomb)) volt)
normalized quantity: 3.456

line: 7 furlong / fortnight
parse: (/ (* 7.0 furlong) fortnight)
normalized quantity: 0.001164145635125 (meter)/(second)
```