```
def          ::= (val variable-name exp)
               | exp
               | (define function-name (formals) exp)
             ⋆ | (record record-name [{field-name}])
               | (use file-name)
               | unit-test

unit-test    ::= (check-expect exp exp)
               | (check-assert exp)
               | (check-error exp)

exp          ::= literal
               | variable-name
               | (set variable-name exp)
               | (if exp exp exp)
               | (while exp exp)
               | (begin {exp})
               | (exp {exp})
               | (let-keyword ({[variable-name exp]}) exp)
               | (lambda (formals) exp)
             ⋆ | (&& {exp}) | (|| {exp})
             ⋆ | (cond {[question-exp answer-exp]})
             ⋆ | (when exp {exp}) | (unless exp {exp})

let-keyword ::= let | let* | letrec

formals      ::= {variable-name}

literal      ::= numeral | #t | #f | 'S-exp | (quote S-exp)

S-exp        ::= symbol-name | numeral | #t | #f | ({S-exp})

numeral      ::= token composed only of digits, possibly prefixed with a plus
                 or minus sign

*-name       ::= token that is not a bracket, a numeral, or one of the "re-
                 served" words shown in typewriter font
```

Tokens are as in Impcore, except that if a quote mark `'` occurs at the beginning of a token, it is a token all by itself; e.g., `'yellow` is two tokens.

Each quoted S-expression is converted to a literal value by the parser. And each `record` definition is expanded to a sequence of true definitions, also by the parser; in other words, a `record` definition is syntactic sugar (Section 1.8 on page 68), as marked by the ⋆. Five forms of conditional expression are also syntactic sugar. All the other forms are handled by the `eval` function.

Figure 2.2: Concrete syntax of μScheme