TRANSCRIPT

A transcript/log of your LLM interaction. It is not necessary to include interactions with the AI about general algorithms or problem decomposition, but do include any interactions in which you ask the LLM to generate or modify code. Include the LLM's answer, and show any further interactions in which you refine the code, attempt to get it to fix errors, etc.

PROJECT START: The Scanner

ME:
Utilizing Racket for the scanner, can we use the first approach?

The scanner runs in its entirety before parsing begins. The scanner is a function (possibly with some auxiliary or sub-functions) that takes in the input file as a single large string, or as a list of strings, one line per string. The scanner function then returns a list of tokens, consisting of data items, each beginning with a label, followed by its value or string representation.

CHATGPT:
Example of a Simple Scanner
racket
Copy code

```
(define (tokenize input-string)
  (define (scan input tokens)
    (cond
      [(empty? input) (reverse tokens)] ; End of input
      [(regexp-match #px"^[a-zA-Z]+" input) ; Identifier
       => (lambda (match)
            (let* ([token (car match)]
                   [rest (substring input (string-length token))])
              (scan rest (cons `(id ,token) tokens))))]
      [(regexp-match #px"^[0-9]+" input) ; Number
       => (lambda (match)
            (let* ([token (car match)]
                   [rest (substring input (string-length token))])
              (scan rest (cons `(num ,token) tokens))))]
      [(regexp-match #px"^\\+" input) ; Plus operator
       => (lambda (match)
            (scan (substring input 1) (cons `(plus "+") tokens)))]
      ;; Add more token types as needed
      [else (error "Unexpected character" input)]))
  (scan input-string '()))

;; Example usage
(define input "(x := y + 23)")
(tokenize input)
```

ME:
I got an error: [else (error "Unexpected character" input)]))

CHATGPT:
Updated tokenize Function

```
#lang racket

(define (tokenize input-string)
  (define (scan input tokens)
    (cond
      ;; Skip whitespace
      [(regexp-match #px"^\s+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match comments (REM followed by any characters until newline)
      [(regexp-match #px"^REM[^\n]*" (string-upcase input))
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match multi-character operators first
      [(regexp-match #px"^:=" input)
       => (lambda (match)
            (let* ([matched ":="]
                   [rest (substring input 2)])
              (scan rest (cons `(assign ,matched) tokens))))]

      ;; Match single-character operators and symbols
      [(regexp-match #px"^[\+\-\*/=<>():]" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input 1)]
                   [token
                    (cond
                      [(string=? matched "+") `(plus "+")]
                      [(string=? matched "-") `(minus "-")]
                      [(string=? matched "*") `(times "*")]
                      [(string=? matched "/") `(divide "/")]
                      [(string=? matched "(") `(lparen "(")]
                      [(string=? matched ")") `(rparen ")")]
                      [(string=? matched ":") `(colon ":")]
                      [(string=? matched "=") `(eq "=")]
                      [(string=? matched "<") `(lt "<")]
                      [(string=? matched ">") `(gt ">")]
                      [else (error "Unknown single-character token"
matched)])])
              (scan rest (cons token tokens))))]

      ;; Match strings (quoted with double quotes)
      [(regexp-match #px"^\"([^\"\n]*)\"" input)
       => (lambda (match)
            (let* ([matched (cadr match)] ; Extract the string without quotes
                   [rest (substring input (+ 2 (string-length matched)))]
                   [token `(string ,matched)])
              (scan rest (cons token tokens))))]

      ;; Match keywords and identifiers (case-insensitive)
      [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
       => (lambda (match)
            (let* ([matched (string-downcase (car match))] ; Convert to
lowercase for consistency
                   [rest (substring input (string-length (car match)))]
                   [token
                    (cond
                      ;; Keywords
                      [(member matched '("def" "enddef" "end" "if" "then"
"endif" "while" "do" "endwhile" "print" "return" "not" "and" "or"))
                       `(keyword ,matched)]
                      ;; Identifiers
                      [else `(id ,(car match))])])
              (scan rest (cons token tokens))))]

      ;; Match integers
      [(regexp-match #px"^[0-9]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(integer ,matched) tokens))))]

      ;; Match real numbers
      [(regexp-match #px"^[0-9]+\.[0-9]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(real ,matched) tokens))))]
      ;; If no patterns match, raise an error
      [else (error "Unexpected character sequence:" input)]))

  (reverse (scan input-string '())))

;; Example usage
(define input "
IF x := y + 23
REM This is a comment
PRINT \"Value of x is:\"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
")

(tokenize input)
```

ME:

```
Error: read-syntax: unknown escape sequence \s in string
```

CHATGPT:

Corrected tokenize Function

```
#lang racket

(define (tokenize input-string)
  (define (scan input tokens)
    (cond
      ;; Skip whitespace (using `[[:space:]]` instead of `\s`)
      [(regexp-match #px"[[:space:]]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match comments (REM followed by any characters until newline)
      [(regexp-match #px"^REM[^\n]*" (string-upcase input))
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match multi-character operators first
      [(regexp-match #px"^:=" input)
       => (lambda (match)
            (let* ([matched ":="]
                   [rest (substring input 2)])
              (scan rest (cons `(assign ,matched) tokens))))]

      ;; Match single-character operators and symbols (properly escaping
special characters)
      [(regexp-match #px"^[\\+\\-\\*/=<>():]" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input 1)]
                   [token
                    (cond
                      [(string=? matched "+") `(plus "+")]
                      [(string=? matched "-") `(minus "-")]
                      [(string=? matched "*") `(times "*")]
                      [(string=? matched "/") `(divide "/")]
                      [(string=? matched "(") `(lparen "(")]
                      [(string=? matched ")") `(rparen ")")]
                      [(string=? matched ":") `(colon ":")]
                      [(string=? matched "=") `(eq "=")]
                      [(string=? matched "<") `(lt "<")]
                      [(string=? matched ">") `(gt ">")]
                      [else (error "Unknown single-character token"
matched)])])
              (scan rest (cons token tokens))))]

      ;; Match strings (quoted with double quotes)
      [(regexp-match #px"^\"([^\"\n]*)\"" input)
       => (lambda (match)
            (let* ([matched (cadr match)] ; Extract the string without quotes
                   [rest (substring input (+ 2 (string-length matched)))]
                   [token `(string ,matched)])
              (scan rest (cons token tokens))))]

      ;; Match keywords and identifiers (case-insensitive)
      [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
       => (lambda (match)
            (let* ([matched (string-downcase (car match))] ; Convert to
lowercase for consistency
                   [rest (substring input (string-length (car match)))]
                   [token
                    (cond
                      ;; Keywords
                      [(member matched '("def" "enddef" "end" "if" "then"
"endif" "while" "do" "endwhile" "print" "return" "not" "and" "or"))
                       `(keyword ,matched)]
                      ;; Identifiers
                      [else `(id ,matched)])])
              (scan rest (cons token tokens))))]

      ;; Match integers
      [(regexp-match #px"^[0-9]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(integer ,matched) tokens))))]

      ;; Match real numbers
      [(regexp-match #px"^[0-9]+\\.[0-9]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(real ,matched) tokens))))]

      ;; If no patterns match, raise an error
      [else (error "Unexpected character sequence:" input)]))

  (reverse (scan input-string '())))

;; Example usage
(define input "
IF x := y + 23
REM This is a comment
PRINT \"Value of x is:\"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
")

(tokenize input)
```

ME:

```
Error: Unexpected character sequence: ""
```

CHATGPT:

Updated tokenize Function with Better Error Handling

racket

Copy code

```
#lang racket

(define (tokenize input-string)
  (define (scan input tokens)
    (cond
      ;; Check if the input is empty or only contains whitespace
      [(string-blank? input)
       (reverse tokens)]

      ;; Skip whitespace (using `[[:space:]]` instead of `\s`)
      [(regexp-match #px"[[:space:]]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match comments (REM followed by any characters until newline)
      [(regexp-match #px"^REM[^\n]*" (string-upcase input))
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match multi-character operators first
      [(regexp-match #px"^:=" input)
       => (lambda (match)
            (let* ([matched ":="]
                   [rest (substring input 2)])
              (scan rest (cons `(assign ,matched) tokens))))]

      ;; Match single-character operators and symbols (properly escaping
special characters)
      [(regexp-match #px"^[\\+\\-\\*/=<>():]" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input 1)]
                   [token
                    (cond
                      [(string=? matched "+") `(plus "+")]
                      [(string=? matched "-") `(minus "-")]
                      [(string=? matched "*") `(times "*")]
                      [(string=? matched "/") `(divide "/")]
                      [(string=? matched "(") `(lparen "(")]
                      [(string=? matched ")") `(rparen ")")]
                      [(string=? matched ":") `(colon ":")]
                      [(string=? matched "=") `(eq "=")]
                      [(string=? matched "<") `(lt "<")]
                      [(string=? matched ">") `(gt ">")]
                      [else (error "Unknown single-character token"
matched)])])
              (scan rest (cons token tokens))))]

      ;; Match strings (quoted with double quotes)
      [(regexp-match #px"^\"([^\"\n]*)\"" input)
       => (lambda (match)
            (let* ([matched (cadr match)] ; Extract the string without quotes
                   [rest (substring input (+ 2 (string-length matched)))]
                   [token `(string ,matched)])
              (scan rest (cons token tokens))))]

      ;; Match keywords and identifiers (case-insensitive)
      [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
       => (lambda (match)
            (let* ([matched (string-downcase (car match))] ; Convert to
lowercase for consistency
                   [rest (substring input (string-length (car match)))]
                   [token
                    (cond
                      ;; Keywords
                      [(member matched '("def" "enddef" "end" "if" "then"
"endif" "while" "do" "endwhile" "print" "return" "not" "and" "or"))
                       `(keyword ,matched)]
                      ;; Identifiers
                      [else `(id ,matched)])])
              (scan rest (cons token tokens))))]

      ;; Match integers
      [(regexp-match #px"^[0-9]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(integer ,matched) tokens))))]

      ;; Match real numbers
      [(regexp-match #px"^[0-9]+\\.[0-9]+" input)
```

```
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(real ,matched) tokens))))]

      ;; If no patterns match, provide a more detailed error message
      [else (error "Unexpected character sequence:" (substring input 0 (min (string-length input) 10)))]))

  (reverse (scan input-string '())))

;; Example usage
(define input "
IF x := y + 23
REM This is a comment
PRINT \"Value of x is:\"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
")

(tokenize input)
```

ME:
Error: string-blank?: unbound identifier in: string-blank?

CHATGPT:
Updated Code:
Here’s the complete code with the custom string-blank? function included:

```
#lang racket

;; Custom function to check if a string is blank (contains only spaces, tabs, newlines, etc.)
(define (string-blank? str)
  (regexp-match? #px"^[[:space:]]*$" str))

(define (tokenize input-string)
  (define (scan input tokens)
    (cond
      ;; Check if the input is empty or only contains whitespace
      [(string-blank? input)
       (reverse tokens)]
      ;; Skip whitespace (using `[[:space:]]` instead of `\s`)
      [(regexp-match #px"[[:space:]]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match comments (REM followed by any characters until newline)
      [(regexp-match #px"^REM[^\n]*" (string-upcase input))
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match multi-character operators first
      [(regexp-match #px"^:=" input)
       => (lambda (match)
            (let* ([matched ":="]
                   [rest (substring input 2)])
              (scan rest (cons `(assign ,matched) tokens))))]

      ;; Match single-character operators and symbols (properly escaping
special characters)
      [(regexp-match #px"^[\\+\\-\\*/=<>():]" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input 1)]
                   [token
                    (cond
                      [(string=? matched "+") `(plus "+")]
                      [(string=? matched "-") `(minus "-")]
                      [(string=? matched "*") `(times "*")]
                      [(string=? matched "/") `(divide "/")]
                      [(string=? matched "(") `(lparen "(")]
                      [(string=? matched ")") `(rparen ")")]
                      [(string=? matched ":") `(colon ":")]
                      [(string=? matched "=") `(eq "=")]
                      [(string=? matched "<") `(lt "<")]
                      [(string=? matched ">") `(gt ">")]
                      [else (error "Unknown single-character token"
matched)])])
              (scan rest (cons token tokens))))]
      ;; Match strings (quoted with double quotes)
      [(regexp-match #px"^\"([^\"\n]*)\"" input)
       => (lambda (match)
            (let* ([matched (cadr match)] ; Extract the string without quotes
                   [rest (substring input (+ 2 (string-length matched)))]
                   [token `(string ,matched)])
              (scan rest (cons token tokens))))]

      ;; Match keywords and identifiers (case-insensitive)
      [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
       => (lambda (match)
            (let* ([matched (string-downcase (car match))] ; Convert to
lowercase for consistency
                   [rest (substring input (string-length (car match)))]
                   [token
                    (cond
                      ;; Keywords
                      [(member matched '("def" "enddef" "end" "if" "then"
"endif" "while" "do" "endwhile" "print" "return" "not" "and" "or"))
                       `(keyword ,matched)]
                      ;; Identifiers
                      [else `(id ,matched)])])
              (scan rest (cons token tokens))))]

      ;; Match integers
      [(regexp-match #px"^[0-9]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(integer ,matched) tokens))))]

      ;; Match real numbers
      [(regexp-match #px"^[0-9]+\\.[0-9]+" input)
       => (lambda (match)
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(real ,matched) tokens))))]

      ;; If no patterns match, provide a more detailed error message
      [else (error "Unexpected character sequence:" (substring input 0 (min
(string-length input) 10)))]))

  (reverse (scan input-string '())))
```

```
;; Example usage
(define input "
IF x := y + 23
REM This is a comment
PRINT \"Value of x is:\"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
")

(tokenize input)
```

ME:

```
Error: Module Language: there can only be one expression in the definitions window in: (define input "\nIF x := y + 23\nREM This is a comment\nPRINT \"Value of x is:\"\nWHILE x < 100 DO\n  x := x + 1\nENDWHILE\nEND\n")
```

CHATGPT:

```
#lang racket

;; Custom function to check if a string is blank (contains only spaces, tabs, newlines, etc.)
(define (string-blank? str)
  (regexp-match? #px"^[[:space:]]*$" str))

(define (tokenize input-string)
  (define (scan input tokens)
    (displayln input)  ;; Print the current input to see what is being processed
    (cond
      ;; Check if the input is empty or only contains whitespace
      [(string-blank? input)
       (reverse tokens)]

      ;; Skip whitespace (using `[[:space:]]` instead of `\s`)
      [(regexp-match #px"[[:space:]]+" input)
       => (lambda (match)
            (displayln "Matched whitespace")
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match comments (REM followed by any characters until newline)
      [(regexp-match #px"^REM[^\n]*" (string-upcase input))
       => (lambda (match)
            (displayln "Matched REM comment")
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest tokens)))]

      ;; Match multi-character operators first
      [(regexp-match #px"^:=" input)
       => (lambda (match)
            (displayln "Matched assignment operator")
            (let* ([matched ":="]
                   [rest (substring input 2)])
              (scan rest (cons `(assign ,matched) tokens))))]

      ;; Match single-character operators and symbols
      [(regexp-match #px"^[\\+\\-\\*/=<>():]" input)
       => (lambda (match)
            (displayln (string-append "Matched single character operator: "
(car match)))
            (let* ([matched (car match)]
                   [rest (substring input 1)]
                   [token
                    (cond
                      [(string=? matched "+") `(plus "+")]
                      [(string=? matched "-") `(minus "-")]
                      [(string=? matched "*") `(times "*")]
                      [(string=? matched "/") `(divide "/")]
                      [(string=? matched "(") `(lparen "(")]
                      [(string=? matched ")") `(rparen ")")]
                      [(string=? matched ":") `(colon ":")]
                      [(string=? matched "=") `(eq "=")]
                      [(string=? matched "<") `(lt "<")]
                      [(string=? matched ">") `(gt ">")]
                      [else (error "Unknown single-character token"
matched)])])
              (scan rest (cons token tokens))))]

      ;; Match strings (quoted with double quotes)
      [(regexp-match #px"^\"([^\"\n]*)\"" input)
       => (lambda (match)
            (displayln "Matched string")
            (let* ([matched (cadr match)] ; Extract the string without quotes
                   [rest (substring input (+ 2 (string-length matched)))]
                   [token `(string ,matched)])
              (scan rest (cons token tokens))))]

      ;; Match keywords and identifiers (case-insensitive)
      [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
       => (lambda (match)
            (displayln "Matched identifier/keyword")
            (let* ([matched (string-downcase (car match))] ; Convert to
lowercase for consistency
                   [rest (substring input (string-length (car match)))]
                   [token
                    (cond
                      ;; Keywords
                      [(member matched '("def" "enddef" "end" "if" "then"
"endif" "while" "do" "endwhile" "print" "return" "not" "and" "or"))
                       `(keyword ,matched)]
                      ;; Identifiers
                      [else `(id ,matched)])])
              (scan rest (cons token tokens))))]

      ;; Match integers
      [(regexp-match #px"^[0-9]+" input)
       => (lambda (match)
            (displayln "Matched integer")
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(integer ,matched) tokens))))]

      ;; Match real numbers
      [(regexp-match #px"^[0-9]+\\.[0-9]+" input)
       => (lambda (match)
            (displayln "Matched real number")
            (let* ([matched (car match)]
                   [rest (substring input (string-length matched))])
              (scan rest (cons `(real ,matched) tokens))))]

      ;; If no patterns match, provide a more detailed error message
      [else (error "Unexpected character sequence:" (substring input 0 (min
(string-length input) 10)))]))

  (reverse (scan input-string '())))

;; Example usage
(define input "
IF x := y + 23
REM This is a comment
PRINT \"Value of x is:\"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
")

(tokenize input)
```

ME:
Interpret this output:

```
IF x := y + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
IF x := y + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
F x := y + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 x := y + 23
REM This is a comment
PRINT "Value of x is:"
```

```
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
x := y + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 := y + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
:= y + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
= y + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 y + 23
REM This is a comment
```

```
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
y + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 + 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
+ 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 23
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
23
```

```
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
3
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace

REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
REM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
EM This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
M This is a comment
PRINT "Value of x is:"
```

```
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
This is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
his is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
is is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
s is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
```

```
Matched whitespace
 is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
is a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
s a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
a comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 comment
PRINT "Value of x is:"
```

```
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
comment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
omment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
mment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
ment
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
ent
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
```

```
Matched whitespace
nt
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
t
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace

PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
PRINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
RINT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
INT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
```

```
END

Matched whitespace
NT "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
T "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 "Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
"Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
Value of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
alue of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
```

```
Matched whitespace
lue of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
ue of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
e of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
of x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
f x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 x is:"
```

```
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
x is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
is:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
s:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
:"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
"
WHILE x < 100 DO
  x := x + 1
```

```
ENDWHILE
END

Matched whitespace

WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
HILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
ILE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
LE x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
E x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 x < 100 DO
  x := x + 1
ENDWHILE
```

```
END

Matched whitespace
x < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 < 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
< 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
 100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
100 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
00 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
0 DO
  x := x + 1
ENDWHILE
END
```

```
Matched whitespace
 DO
  x := x + 1
ENDWHILE
END

Matched whitespace
DO
  x := x + 1
ENDWHILE
END

Matched whitespace
  x := x + 1
ENDWHILE
END

Matched whitespace
x := x + 1
ENDWHILE
END

Matched whitespace
 := x + 1
ENDWHILE
END

Matched whitespace
:= x + 1
ENDWHILE
END

Matched whitespace
= x + 1
ENDWHILE
END

Matched whitespace
 x + 1
ENDWHILE
END

Matched whitespace
x + 1
```

```
ENDWHILE
END

Matched whitespace
 + 1
ENDWHILE
END

Matched whitespace
+ 1
ENDWHILE
END

Matched whitespace
 1
ENDWHILE
END

Matched whitespace
1
ENDWHILE
END

Matched whitespace

ENDWHILE
END

Matched whitespace
ENDWHILE
END

Matched whitespace
NDWHILE
END

Matched whitespace
DWHILE
END

Matched whitespace
WHILE
END
```

```
Matched whitespace
HILE
END

Matched whitespace
ILE
END

Matched whitespace
LE
END

Matched whitespace
E
END

Matched whitespace

END

Matched whitespace
END

Matched whitespace
ND

Matched whitespace
D

Matched whitespace
```

CHATGPT:
It seems like your scanner is primarily matching whitespace and not properly tokenizing the input as expected. The issue could be due to how the regular expressions are structured or how the scanner is processing the input.

From the logs you shared, the scanner repeatedly finds "whitespace" and progresses one character at a time without correctly identifying tokens like IF, x, :=, and so on.

Here are some suggestions to help you fix this:

1. Whitespace Handling

It looks like the whitespace matcher is being overly aggressive, matching almost everything and leaving no room for other token types to be matched. You should adjust the order in which your cond checks are evaluated.

2. Matching Full Words First
Your current code might be matching individual characters (like I, F, etc.) instead of full words (IF, x, :=, etc.). To fix this, try matching multi-character tokens (like IF, PRINT, or :=) before matching individual characters.

3. Reordering the Scanner Logic
Reorder the cond clauses so that longer, multi-character tokens (e.g., keywords like IF, PRINT, operators like :=) are matched before shorter, single-character tokens and whitespace.

Updated Tokenizer with Reordered Matching Logic:

```
#lang racket

;; Custom function to check if a string is blank (contains only spaces, tabs,
newlines, etc.)
(define (string-blank? str)
  (regexp-match? #px"^[[:space:]]*$" str))

;; Custom function to check if a string is empty
(define (string-empty? str)
  (string=? str ""))

(define (tokenize input-string)
  (define (scan input tokens)
    ;; Check if input is empty or blank
    (if (or (string-empty? input) (string-blank? input))
        (reverse tokens) ; Return the accumulated tokens if the input is empty
        (cond
          ;; Match keywords and identifiers (case-insensitive) first
          [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
           => (lambda (match)
                (let* ([matched (string-downcase (car match))] ; Convert to
lowercase for consistency
                       [rest (substring input (string-length (car match)))]
                       [token
                        (cond
                          ;; Keywords
                          [(member matched '("def" "enddef" "end" "if" "then"
"endif" "while" "do" "endwhile" "print" "return" "not" "and" "or"))
                           `(keyword ,matched)]
                          ;; Identifiers
                          [else `(id ,matched)])])
                  (scan rest (cons token tokens))))]

          ;; Match multi-character operators like `:=`
          [(regexp-match #px"^:=" input)
           => (lambda (match)
                (let* ([matched ":="]
                       [rest (substring input 2)])
                  (scan rest (cons `(assign ,matched) tokens))))]

          ;; Match single-character operators and symbols
          [(regexp-match #px"^[\\+\\-\\*/=<>():]" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input 1)]
                       [token
                        (cond
                          [(string=? matched "+") `(plus "+")]
                          [(string=? matched "-") `(minus "-")]
                          [(string=? matched "*") `(times "*")]
                          [(string=? matched "/") `(divide "/")]
                          [(string=? matched "(") `(lparen "(")]
                          [(string=? matched ")") `(rparen ")")]
                          [(string=? matched ":") `(colon ":")]
                          [(string=? matched "=") `(eq "=")]
                          [(string=? matched "<") `(lt "<")]
                          [(string=? matched ">") `(gt ">")]
                          [else (error "Unknown single-character token"
matched)])])
                  (scan rest (cons token tokens))))]

          ;; Match numbers (integer or real)
          [(regexp-match #px"^[0-9]+\\.[0-9]+" input) ;; Match real numbers
first
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest (cons `(real ,matched) tokens))))]

          [(regexp-match #px"^[0-9]+" input) ;; Match integers next
          => (lambda (match)
               (let* ([matched (car match)]
                      [rest (substring input (string-length matched))])
                 (scan rest (cons `(integer ,matched) tokens))))]

          ;; Match strings (quoted with double quotes)
          [(regexp-match #px"^\"([^\"\n]*)\"" input)
           => (lambda (match)
                (let* ([matched (cadr match)] ; Extract the string without
quotes
                       [rest (substring input (+ 2 (string-length matched)))]
                       [token `(string ,matched)])
                  (scan rest (cons token tokens))))]

          ;; Skip whitespace but ensure it's handled after other matches
          [(regexp-match #px"[[:space:]]+" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest tokens)))]

          ;; Handle comments (starting with REM)
          [(regexp-match #px"^REM[^\n]*" (string-upcase input))
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest tokens)))]

          ;; If no patterns match, provide a detailed error message
          [else (error "Unexpected character sequence: " (substring input 0
(min (string-length input) 10)))])))

  ;; Reverse the list of tokens and return
  (reverse (scan input-string '())))

;; Example usage
(define input "
IF x := y + 23
REM This is a comment
PRINT \"Value of x is:\"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
```

```
")

(tokenize input)
```

MID-PROJECT: The Parser

Me:
The parser The parser’s job is to determine whether the tokens follow the grammar, and to build up a parse tree representing the abstract structure of the program. In this grammar, a program is a list of statements. A statement can be represented as a list of tokens, with a suitable label at the head of the list to indicate its type. For example, starting with '((ID "X") (assign-op ":=") (ID "Y") (add-op "+") (Integer 1)) we go through the grammar and eventually get a somewhat more complex list: ((STMT (ID "x") (assign-op ":=") (expr (and-expr (not-expr (compare-expr (add-expr (mult-expr (negate-expr (value (id "Y")))) "+" (add-expr (negate-expr (value (constant (integer 1))))))))))))) What on earth is all that? That’s what we get from working down our grammar—each function identifies what its component should be, and calls the appropriate function; when the function returns, it’s labeled what it found: ((STMT (ID "x") (assign-op ":=") (expr (and-expr (not-expr (compare-expr (add-expr (mult-expr (negate-expr (value (id "Y")))) “+" (add-expr (negate-expr (value (constant (integer 1))))))))))))) (full cascade of closing parens not shown, to save space) But after going through all that, we’ve verified that “X := Y + 1” is in fact a valid statement in this grammar. If every statement in the grammar is valid, then we’ll have a list of statements—a program

CHATGPT:
Parser Implementation Outline
Here’s an outline for how each part of the parser might look in Racket:

racket
Copy code
```
#lang racket

;; Helper function to match and consume a token
(define (match tokens type)
  (if (and (pair? tokens) (eq? (car (car tokens)) type))
      (values (cadr (car tokens)) (cdr tokens))
      (error "Unexpected token type")))

;; Parser entry function
(define (parse-program tokens)
  (define (parse-statements tokens)
    (if (null? tokens)
        '()
        (let-values ([(stmt rest) (parse-stmt tokens)])
          (cons stmt (parse-statements rest)))))
  (parse-statements tokens))

;; Parse a statement
(define (parse-stmt tokens)
  (match (car tokens)
    ;; Assignment Statement
    [('ID _)
     (define-values (var rest) (match tokens 'ID))
     (define-values (op rest) (match rest 'assign-op))
     (define-values (expr rest) (parse-expr rest))
     (values `(STMT (ID ,var) (assign-op ,op) (expr ,expr)) rest)]

    ;; Other statement types (e.g., IF, WHILE, PRINT, etc.) would follow
similar patterns
    ))

;; Parse an expression
(define (parse-expr tokens)
  (let-values ([(left rest) (parse-add-expr tokens)])
    (if (and (pair? rest) (equal? (car (car rest)) 'eq))
        (let-values ([(op _) (match rest 'eq)]
                     [(right rest) (parse-add-expr rest)])
          (values `(expr (compare-expr ,left ,op ,right)) rest))
        (values `(expr ,left) rest))))

;; Parse an addition expression
(define (parse-add-expr tokens)
  (let-values ([(left rest) (parse-mult-expr tokens)])
    (if (and (pair? rest) (equal? (car (car rest)) 'add-op))
        (let-values ([(op _) (match rest 'add-op)]
                     [(right rest) (parse-mult-expr rest)])
          (values `(add-expr ,left ,op ,right) rest))
        (values `(add-expr ,left) rest))))

;; Parse a multiplication expression
(define (parse-mult-expr tokens)
  (let-values ([(left rest) (parse-negate-expr tokens)])
    (if (and (pair? rest) (equal? (car (car rest)) 'times))
        (let-values ([(op _) (match rest 'times)]
                     [(right rest) (parse-negate-expr rest)])
          (values `(mult-expr ,left ,op ,right) rest))
        (values `(mult-expr ,left) rest))))

;; Parse a negation expression
(define (parse-negate-expr tokens)
  (if (and (pair? tokens) (equal? (car (car tokens)) 'minus))
      (let-values ([(op _) (match tokens 'minus)]
                   [(expr rest) (parse-value tokens)])
        (values `(negate-expr ,op ,expr) rest))
      (parse-value tokens)))

;; Parse a value (ID, constant, or expression in parentheses)
(define (parse-value tokens)
  (match (car tokens)
    [('ID _)
     (define-values (id rest) (match tokens 'ID))
     (values `(id ,id) rest)]
    [('integer _)
     (define-values (num rest) (match tokens 'integer))
     (values `(constant (integer ,num)) rest)]
    [('lparen _)
     (define-values (_ rest) (match tokens 'lparen))
     (define-values (expr rest) (parse-expr rest))
     (define-values (_ rest) (match rest 'rparen))
     (values `(value ,expr) rest)]
    (else (error "Unexpected value type"))))
```

ME:

for the given scanner code, can you implement the parser portion? #lang racket

```
;; Custom function to check if a string is blank (contains only spaces, tabs, newlines, etc.)
(define (string-blank? str)
  (regexp-match? #px"^[[:space:]]*$" str))

;; Custom function to check if a string is empty
(define (string-empty? str)
  (string=? str ""))

(define (tokenize input-string)
  (define (scan input tokens)
    ;; Check if input is empty or blank
    (if (or (string-empty? input) (string-blank? input))
        (reverse tokens) ; Return the accumulated tokens if the input is empty
        (cond
          ;; Match keywords and identifiers (case-insensitive) first
          [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
           => (lambda (match)
                (let* ([matched (string-downcase (car match))] ; Convert to lowercase for consistency
                       [rest (substring input (string-length (car match)))]
                       [token
                        (cond
                          ;; Keywords
                          [(member matched '("def" "enddef" "end" "if" "then" "endif" "while" "do" "endwhile" "print" "return" "not" "and" "or"))
                           (keyword ,matched)]
                          ;; Identifiers
                          [else (id ,matched)])])
                  (scan rest (cons token tokens))))]

          ;; Match multi-character operators like :=
          [(regexp-match #px"^:=" input)
           => (lambda (match)
                (let* ([matched ":="]
                       [rest (substring input 2)])
                  (scan rest (cons (assign ,matched) tokens))))]

          ;; Match single-character operators and symbols
          [(regexp-match #px"^[\\+\\-\\*/=<>():]" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input 1)]
                       [token
                        (cond
                          [(string=? matched "+") (plus "+")]
                          [(string=? matched "-") (minus "-")]
                          [(string=? matched "*") (times "*")]
                          [(string=? matched "/") (divide "/")]
                          [(string=? matched "(") (lparen "(")]
                          [(string=? matched ")") (rparen ")")]
                          [(string=? matched ":") (colon ":")]
                          [(string=? matched "=") (eq "=")]
                          [(string=? matched "<") (lt "<")]
                          [(string=? matched ">") (gt ">")]
                          [else (error "Unknown single-character token" matched)])])
                  (scan rest (cons token tokens))))]
          ;; Match numbers (integer or real)
          [(regexp-match #px"^[0-9]+\\.[0-9]+" input) ;; Match real numbers
first
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest (cons (real ,matched) tokens))))]

          [(regexp-match #px"^[0-9]+" input) ;; Match integers next
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest (cons (integer ,matched) tokens))))]

          ;; Match strings (quoted with double quotes)
          [(regexp-match #px"^\"([^\"\n]*)\"" input)
           => (lambda (match)
                (let* ([matched (cadr match)] ; Extract the string without
quotes
                       [rest (substring input (+ 2 (string-length matched)))]
                       [token (string ,matched)])
                  (scan rest (cons token tokens))))]

          ;; Skip whitespace but ensure it's handled after other matches
          [(regexp-match #px"[[:space:]]+" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest tokens)))]

          ;; Handle comments (starting with REM)
          [(regexp-match #px"^REM[^\n]*" (string-upcase input))
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest tokens)))]

          ;; If no patterns match, provide a detailed error message
          [else (error "Unexpected character sequence: " (substring input 0
(min (string-length input) 10)))])))

  ;; Reverse the list of tokens and return
  (reverse (scan input-string '())))
```

```
;; Example usage
(define input "
IF x := y + 23
REM This is a comment
PRINT \"Value of x is:\"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
")

(tokenize input)
```

CHATGPT:
Parser Code
Here’s a possible implementation of the parser:

racket
Copy code

```
#lang racket

;; Define the main function to parse a program
(define (parse-program tokens)
  (define (parse-statements tokens)
    (if (null? tokens)
        '()
        (let-values ([(stmt rest) (parse-stmt tokens)])
          (cons stmt (parse-statements rest)))))
  (parse-statements tokens))

;; Parse a statement
(define (parse-stmt tokens)
  (match (car tokens)
    ;; Assignment Statement
    [(list 'id var)
     (define-values (op rest) (match (cdr tokens) '(assign _)))
     (define-values (expr rest) (parse-expr rest))
     (values `(STMT (id ,var) (assign ,op) (expr ,expr)) rest)]

    ;; Print Statement
    [(list 'print)
     (define-values (string rest) (parse-expr (cdr tokens)))
     (values `(STMT (print ,string)) rest)]
    ;; While Loop
    [(list 'while)
     (define-values (condition rest) (parse-expr (cdr tokens)))
     (define-values (do rest) (parse-stmt rest))
     (values `(STMT (while ,condition ,do)) rest)]

    ;; Additional types (IF, function definitions) can be added similarly
    ))

;; Parse an expression
(define (parse-expr tokens)
  (let-values ([(left rest) (parse-add-expr tokens)])
    (if (and (pair? rest) (equal? (car (car rest)) 'eq))
        (let-values ([(op _) (match rest 'eq)]
                     [(right rest) (parse-add-expr rest)])
          (values `(expr (compare-expr ,left ,op ,right)) rest))
        (values `(expr ,left) rest))))

;; Parse addition/subtraction expressions
(define (parse-add-expr tokens)
  (let-values ([(left rest) (parse-mult-expr tokens)])
    (if (and (pair? rest) (equal? (car (car rest)) 'add-op))
        (let-values ([(op _) (match rest 'add-op)]
                     [(right rest) (parse-mult-expr rest)])
          (values `(add-expr ,left ,op ,right) rest))
        (values `(add-expr ,left) rest))))

;; Parse multiplication/division expressions
(define (parse-mult-expr tokens)
  (let-values ([(left rest) (parse-negate-expr tokens)])
    (if (and (pair? rest) (equal? (car (car rest)) 'times))
        (let-values ([(op _) (match rest 'times)]
                     [(right rest) (parse-negate-expr rest)])
          (values `(mult-expr ,left ,op ,right) rest))
        (values `(mult-expr ,left) rest))))

;; Parse a negation expression or single value
(define (parse-negate-expr tokens)
  (if (and (pair? tokens) (equal? (car (car tokens)) 'minus))
      (let-values ([(op _) (match tokens 'minus)]
                   [(expr rest) (parse-value tokens)])
        (values `(negate-expr ,op ,expr) rest))
      (parse-value tokens)))
```

```
;; Parse a value (ID, constant, or expression in parentheses)
(define (parse-value tokens)
  (match (car tokens)
    [(list 'id val)
     (values `(id ,val) (cdr tokens))]
    [(list 'integer val)
     (values `(constant (integer ,val)) (cdr tokens))]
    [(list 'lparen _)
     (define-values (_ rest) (match tokens 'lparen))
     (define-values (expr rest) (parse-expr rest))
     (define-values (_ rest) (match rest 'rparen))
     (values `(value ,expr) rest)]
    (else (error "Unexpected value type"))))

;; Testing with a simple program

(define input "
IF x := y + 23
PRINT \"Value of x is:\"
WHILE x < 100 DO
  x := x + 1
ENDWHILE
END
")

;; Run the tokenizer and parse the resulting tokens
(define tokens (tokenize input))
(define parse-tree (parse-program tokens))

;; Display the resulting parse tree
(displayln parse-tree)
```

ME:

```
Error: id: unbound identifier in: id
```

CHATGPT:

```
#lang racket

;; Custom function to check if a string is blank (contains only spaces, tabs, newlines, etc.)
(define (string-blank? str)
  (regexp-match? #px"^[[:space:]]*$" str))
```

```
;; Custom function to check if a string is empty
(define (string-empty? str)
  (string=? str ""))

;; Tokenizer function
(define (tokenize input-string)
  (define (scan input tokens)
    (if (or (string-empty? input) (string-blank? input))
        (reverse tokens)
        (cond
          [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
           => (lambda (match)
                (let* ([matched (string-downcase (car match))]
                       [rest (substring input (string-length (car match)))])
                  (scan rest (cons `(id ,matched) tokens))))]
          [(regexp-match #px"^:=" input)
           => (lambda (match)
                (let* ([rest (substring input 2)])
                  (scan rest (cons `(assign ":=") tokens))))]
          [(regexp-match #px"^[\\+\\-\\*/=<>():;]" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input 1)]
                       [token
                        (cond
                          [(string=? matched "+") `(plus "+")]
                          [(string=? matched "-") `(minus "-")]
                          [(string=? matched "*") `(times "*")]
                          [(string=? matched "/") `(divide "/")]
                          [(string=? matched "(") `(lparen "(")]
                          [(string=? matched ")") `(rparen ")")]
                          [(string=? matched ":") `(colon ":")]
                          [(string=? matched ";") `(semicolon ";")]
                          [(string=? matched "=") `(eq "=")]
                          [(string=? matched "<") `(lt "<")]
                          [(string=? matched ">") `(gt ">")]
                          [else (error "Unknown single-character token"
matched)])])
                  (scan rest (cons token tokens))))]
          [(regexp-match #px"^[0-9]+\\.[0-9]+" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest (cons `(real ,matched) tokens))))]
          [(regexp-match #px"^[0-9]+" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest (cons `(integer ,matched) tokens))))]
          [(regexp-match #px"^\"([^\"\n]*)\"" input)
           => (lambda (match)
                (let* ([matched (cadr match)]
                       [rest (substring input (+ 2 (string-length matched)))])
                  (scan rest (cons `(string ,matched) tokens))))]
          [(regexp-match #px"[[:space:]]+" input)
           => (lambda (match)
                (let* ([rest (substring input (string-length (car match)))])
                  (scan rest tokens)))]
          [(regexp-match #px"^REM[^\n]*" (string-upcase input))
           => (lambda (match)
                (let* ([rest (substring input (string-length (car match)))])
                  (scan rest tokens)))]
          [else (error "Unexpected character sequence: " (substring input 0
(min (string-length input) 10)))])))
  (reverse (scan input-string '())))

;; Main program parser function
(define (parse-program tokens)
  (define (parse-statements tokens)
    (if (null? tokens)
        '()
        (let* ([stmt-and-rest (parse-stmt tokens)]
               [stmt (first stmt-and-rest)]
               [rest1 (second stmt-and-rest)])
          (cons stmt (parse-statements rest1)))))
  (parse-statements tokens))

;; Parse a statement
(define (parse-stmt tokens)
  (cond
    [(and (pair? tokens) (equal? (caar tokens) 'id) (equal? (caar (cdr
tokens)) 'assign))
     (let* ([var (cadar tokens)]
            [rest-assignment (cddr tokens)]
            [expr-and-rest (parse-expr rest-assignment)]
            [expr (car expr-and-rest)]
            [rest-after-expr (cdr expr-and-rest)])
       (list `(STMT (id ,var) (assign ":=") (expr ,expr)) rest-after-expr))]
    [(and (pair? tokens) (equal? (caar tokens) 'print))
     (let* ([rest-print (cdr tokens)]
            [expr-and-rest (parse-expr rest-print)]
            [expr (car expr-and-rest)]
            [rest-after-print (cdr expr-and-rest)])
       (list `(STMT (print ,expr)) rest-after-print))]
    [(and (pair? tokens) (equal? (caar tokens) 'while))
     (let* ([rest-while (cdr tokens)]
            [condition-and-rest (parse-expr rest-while)]
            [condition (car condition-and-rest)]
            [rest-after-condition (cdr condition-and-rest)]
            [do-stmt-and-rest (parse-stmt rest-after-condition)]
            [do-stmt (car do-stmt-and-rest)]
            [rest-after-do (cdr do-stmt-and-rest)])
       (list `(STMT (while ,condition ,do-stmt)) rest-after-do))]
    [else (error "Unexpected statement format or unrecognized token sequence"
tokens)]))

;; Parse expressions and components
(define (parse-expr tokens)
  (let* ([left-and-rest (parse-add-expr tokens)]
         [left (car left-and-rest)]
         [rest (cdr left-and-rest)])
    (if (and (pair? rest) (equal? (caar rest) 'eq))
        (let* ([op (car rest)]
               [rest-after-eq (cdr rest)]
               [right-and-rest (parse-add-expr rest-after-eq)]
               [right (car right-and-rest)]
               [rest-after-right (cdr right-and-rest)])
          (list `(expr (compare-expr ,left ,op ,right)) rest-after-right))
        (list `(expr ,left) rest))))

(define (parse-add-expr tokens)
  (let* ([left-and-rest (parse-mult-expr tokens)]
         [left (car left-and-rest)]
         [rest (cdr left-and-rest)])
    (if (and (pair? rest) (equal? (caar rest) 'plus))
        (let* ([op (car rest)]
               [rest-after-plus (cdr rest)]
               [right-and-rest (parse-mult-expr rest-after-plus)]
               [right (car right-and-rest)]
               [rest-after-right (cdr right-and-rest)])
          (list `(add-expr ,left ,op ,right) rest-after-right))
        (list `(add-expr ,left) rest))))
```

```
(define (parse-mult-expr tokens)
  (let* ([left-and-rest (parse-negate-expr tokens)]
         [left (car left-and-rest)]
         [rest (cdr left-and-rest)])
    (if (and (pair? rest) (equal? (caar rest) 'times))
        (let* ([op (car rest)]
               [rest-after-times (cdr rest)]
               [right-and-rest (parse-negate-expr rest-after-times)]
               [right (car right-and-rest)]
               [rest-after-right (cdr right-and-rest)])
          (list `(mult-expr ,left ,op ,right) rest-after-right))
        (list `(mult-expr ,left) rest))))

(define (parse-negate-expr tokens)
  (if (and (pair? tokens) (equal? (caar tokens) 'minus))
      (let* ([op (car tokens)]
             [rest-after-minus (cdr tokens)]
             [expr-and-rest (parse-value rest-after-minus)]
             [expr (car expr-and-rest)]
             [rest-after-expr (cdr expr-and-rest)])
        (list `(negate-expr ,op ,expr) rest-after-expr))
      (parse-value tokens)))

(define (parse-value tokens)
  (match (car tokens)
    [(list 'id val)
     (list `(id ,val) (cdr tokens))]
    [(list 'integer val)
     (list `(constant (integer ,val)) (cdr tokens))]
    [(list 'lparen _)
     (let* ([rest-after-lparen (cdr tokens)]
            [expr-and-rest (parse-expr rest-after-lparen)]
            [expr (car expr-and-rest)]
            [rest-after-expr (cdr expr-and-rest)]
            [rest-after-rparen (cdr rest-after-expr)])
       (list `(value ,expr) rest-after-rparen))]
    (else (error "Unexpected value type"))))

;; Test Input
(define test-input "
REM example comments
x := 5
y := x + 1 : z := (y - 2) * 3
A := somefunction(z)
PRINT (z * 2)
PRINT \"A = \" ; A
END
DEF somefunction(a)
  WHILE a <= 0 DO
      a := a + 10
  ENDWHILE : REM just some fiddling around to test the language
  IF a > 5 then
    a := a + 2
    RETURN a * 3
  ENDIF
  RETURN a : REM else clause
ENDDEF
")

;; Tokenize the input
(define tokens (tokenize test-input))
(displayln "Tokens:")
(displayln tokens)

;; Parse the program
(define parse-tree (parse-program tokens))
(displayln "\nParse Tree:")
(displayln parse-tree)
```

MID-PROJECT Note:

From here, CHATGPT looped into an “unbound identifier” error cycle for hours similar to the one above this text. At this point I was frustrated with the lack of progress, thus I decided to work on the parser from here by myself. Eventually, after I got a mostly functioning parser, I decided to use CHATGPT to see if it could clean up the rest of what I already had.

Me:
Please help me correct the Parser of my source code:

```
#lang racket

;; ---------------------------------------------------------------------
;; Tokenizer
;; -------------------------------------------------------------------

;; Custom function to check if a string is blank (contains only spaces, tabs,
newlines, etc.)
(define (string-blank? str)
  (regexp-match? #px"^[[:space:]]*$" str))

;; Custom function to check if a string is empty
(define (string-empty? str)
  (string=? str ""))

(define (tokenize input-string)
  (define (scan input tokens)
    ;; Check if input is empty or blank
    (if (or (string-empty? input) (string-blank? input))
        (reverse tokens) ; Return the accumulated tokens if the input is empty
        (cond
          ;; Handle comments (starting with REM) - needs to be before ID
matching
          [(regexp-match #px"^(REM)[^\n]*" (string-upcase input))
           => (lambda (match)
                (let* ([matched (car match)] ; "REM..."
                       [rest (substring input (string-length matched))])
                  (scan rest (cons (remark ,(substring matched 3)) tokens))))]

          ;; Match keywords and identifiers (case-insensitive) first
          [(regexp-match #px"^[A-Za-z][A-Za-z0-9]*" input)
           => (lambda (match)
                (let* ([matched (substring input 0 (string-length (car
match)))]
                       [rest (substring input (string-length matched))]
                       [lower-matched (string-downcase matched)] ; Convert to
lowercase for consistency
                       [token
                        (cond
                          ;; Keywords
                          [(member lower-matched '("def" "enddef" "end" "if"
"then" "endif"
                                                 "while" "do" "endwhile"
"print" "return"
                                                 "not" "and" "or"))
                           (keyword ,lower-matched)]
                          ;; Identifiers
                          [else (id ,matched)])])
                  (scan rest (cons token tokens))))]

          ;; Match multi-character operators like :=, <=, >=, <>, ><
          [(regexp-match #px"^(<>|><|<=|>=|:=)" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))]
                       [token
                        (cond
                          [(string=? matched ":=") (assign ,matched)]
                          [(member matched '("<>" "><" "<=" ">=")) (operator
,matched)]
                          [else (error "Unknown multi-character operator"
matched)])])
                  (scan rest (cons token tokens))))]

          ;; Match single-character operators and symbols
          [(regexp-match #px"^[+*/=<>():;,-]" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input 1)]
                       [token
                        (cond
                          [(string=? matched "+") (plus "+")]
                          [(string=? matched "-") (minus "-")]
                          [(string=? matched "*") (times "*")]
                          [(string=? matched "/") (divide "/")]
                          [(string=? matched "(") (lparen "(")]
                          [(string=? matched ")") (rparen ")")]
                          [(string=? matched ":") (colon ":")]
                          [(string=? matched "=") (operator "=")]
                          [(string=? matched "<") (operator "<")]
                          [(string=? matched ">") (operator ">")]
                          [(string=? matched ";") (semicolon ";")]
                          [(string=? matched ",") (comma ",")]
                          [else (error "Unknown single-character token"
matched)])])
                  (scan rest (cons token tokens))))]

          ;; Match numbers (integer or real)
          [(regexp-match #px"^[0-9]+\\.[0-9]+" input) ;; Match real numbers
first
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest (cons (real ,matched) tokens))))]

          [(regexp-match #px"^[0-9]+" input) ;; Match integers next
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest (cons (integer ,matched) tokens))))]

          ;; Match strings (quoted with double quotes)
          [(regexp-match #px"^\"([^\"]*)\"" input)
           => (lambda (match)
                (let* ([matched (cadr match)] ; Extract the string without
quotes
                       [rest (substring input (+ 2 (string-length matched)))]
                       [token (string ,matched)])
                  (scan rest (cons token tokens))))]

          ;; Skip whitespace but ensure it's handled after other matches
          [(regexp-match #px"^[[:space:]]+" input)
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  (scan rest tokens)))]

          ;; If no patterns match, provide a detailed error message
          [else (error "Unexpected character sequence: "
                       (substring input 0 (min (string-length input) 10)))])))

    ;; Start scanning
    (scan input-string '()))

;; ----------------------------------------------------------------------
;; Parser
;; ----------------------------------------------------------------------

;; Parser utility functions
(define (tokens-empty? tokens)
  (null? tokens))

(define (current-token tokens)
  (if (tokens-empty? tokens)
      (error "Unexpected end of input")
      (car tokens)))
```

```
(define (next-tokens tokens)
  (if (tokens-empty? tokens)
      (error "Unexpected end of input")
      (cdr tokens)))

;; Match a token of a specific type and value (case-insensitive for keywords)
(define (match-token tokens expected-type expected-value)
  (let ((token (current-token tokens)))
    (if (and (eq? (car token) expected-type)
             (or (not (string? expected-value))
                 (string-ci=? (cadr token) expected-value)))
        (next-tokens tokens)
        (error (format "Expected ~a '~a', got '~a'"
                       expected-type expected-value (cadr token))))))

;; Match a token of a specific type, returning the token and remaining tokens
(define (match-token-type tokens expected-type)
  (let ((token (current-token tokens)))
    (if (eq? (car token) expected-type)
        (values token (next-tokens tokens))
        (error (format "Expected token of type ~a, got ~a" expected-type (car
token))))))

;; Peek at the next token without consuming it
(define (peek-token tokens)
  (if (tokens-empty? tokens)
      '()
      (current-token tokens)))

;; Parse <Lines>
(define (parse-lines tokens)
  (let-values (((statements tokens-after-statements) (parse-statements
tokens)))
    (if (tokens-empty? tokens-after-statements)
        statements
        (begin
          (displayln "Remaining tokens:")
          (pretty-print tokens-after-statements)
          (error "Unexpected tokens after parsing lines")))))

;; List of end-of-block keywords
(define end-keywords '("endif" "enddef" "endwhile" "end"))
```

```
;; Parse <Statements>
(define (parse-statements tokens)
  (let loop ((tokens tokens)
             (statements '()))
    (if (tokens-empty? tokens)
        (values (reverse statements) tokens)
        (let ((token (current-token tokens)))
          (if (and (eq? (car token) 'keyword)
                   (member (string-downcase (cadr token)) end-keywords))
              ;; End of statements; return the accumulated statements
              (values (reverse statements) tokens)
              ;; Else, parse the next statement
              (let-values (((statement tokens-after-statement)
(parse-statement tokens)))
                (let ((tokens tokens-after-statement))
                  (if (not (tokens-empty? tokens))
                      (let ((token (current-token tokens)))
                        (if (eq? (car token) 'colon)
                            ;; If a colon is encountered, consume it and
continue parsing
                            (loop (next-tokens tokens) (cons statement
statements))
                            ;; No colon, continue parsing with current tokens
                            (loop tokens (cons statement statements))))
                      ;; No more tokens, return the statements
                      (values (reverse (cons statement statements))
tokens))))))))

;; Parse <Statement>
(define (parse-statement tokens)
  (let ((token (current-token tokens)))
    (cond
      ;; DEF ID ( <ID List> ) <Statements> ENDDEF
      [(and (eq? (car token) 'keyword) (string-ci=? (cadr token) "def"))
       (let ((tokens-after-def (next-tokens tokens)))
         (let-values (((id-token tokens-after-id) (match-token-type
tokens-after-def 'id)))
           (let ((tokens-after-lparen (match-token tokens-after-id 'lparen
"(")))
             (let-values (((id-list tokens-after-id-list) (parse-id-list
tokens-after-lparen)))
               (let ((tokens-after-rparen (match-token tokens-after-id-list
'rparen ")")))
```

```
                 (let-values (((body-statements tokens-after-body)
(parse-statements tokens-after-rparen)))
                   (let ((tokens-after-enddef (match-token tokens-after-body
'keyword "enddef")))
                     (values (list 'DEF id-token id-list body-statements)
                             tokens-after-enddef))))))))]

      ;; ENDDEF
      [(and (eq? (car token) 'keyword) (string-ci=? (cadr token) "enddef"))
       (values (list 'ENDDEF) (next-tokens tokens))]

      ;; END
      [(and (eq? (car token) 'keyword) (string-ci=? (cadr token) "end"))
       (values (list 'END) (next-tokens tokens))]

      ;; IF <Expression> THEN <Statements> ENDIF
      [(and (eq? (car token) 'keyword) (string-ci=? (cadr token) "if"))
       (let ((tokens-after-if (next-tokens tokens)))
         (let-values (((expr tokens-after-expr) (parse-expression
tokens-after-if)))
           (let ((tokens-after-then (match-token tokens-after-expr 'keyword
"then")))
             (let-values (((statements tokens-after-statements)
(parse-statements tokens-after-then)))
               (let ((tokens-after-endif (match-token tokens-after-statements
'keyword "endif")))
                 (values (list 'IF expr statements)
                         tokens-after-endif))))))]

      ;; WHILE <Expression> DO <Statements> ENDWHILE
      [(and (eq? (car token) 'keyword) (string-ci=? (cadr token) "while"))
       (let ((tokens-after-while (next-tokens tokens)))
         (let-values (((expr tokens-after-expr) (parse-expression
tokens-after-while)))
           (let ((tokens-after-do (match-token tokens-after-expr 'keyword
"do")))
             (let-values (((statements tokens-after-statements)
(parse-statements tokens-after-do)))
               (let ((tokens-after-endwhile (match-token
tokens-after-statements 'keyword "endwhile")))
                 (values (list 'WHILE expr statements)
                         tokens-after-endwhile))))))]

      ;; PRINT <Print List>
      [(and (eq? (car token) 'keyword) (string-ci=? (cadr token) "print"))
       (let ((tokens-after-print (next-tokens tokens)))
         (let-values (((print-list tokens-after-print-list) (parse-print-list
tokens-after-print)))
           (values (list 'PRINT print-list) tokens-after-print-list)))]

      ;; RETURN <Expression>
      [(and (eq? (car token) 'keyword) (string-ci=? (cadr token) "return"))
       (let ((tokens-after-return (next-tokens tokens)))
         (let-values (((expr tokens-after-expr) (parse-expression
tokens-after-return)))
           (values (list 'RETURN expr) tokens-after-expr)))]

      ;; REMARK
      [(eq? (car token) 'remark)
       (values (list 'REMARK (cadr token)) (next-tokens tokens))]

      ;; ID ':=' <Expression>
      ;; ID '(' <Expression List> ')'
      [(eq? (car token) 'id)
       (let ((tokens-after-id (next-tokens tokens))
             (id-token token))
         (if (not (tokens-empty? tokens-after-id))
             (let ((next-token (current-token tokens-after-id)))
               (cond
                 ;; ID ':=' <Expression>
                 [(eq? (car next-token) 'assign)
                  (let ((tokens-after-assign (next-tokens tokens-after-id)))
                    (let-values (((expr tokens-after-expr) (parse-expression
tokens-after-assign)))
                      (values (list 'ASSIGN id-token expr)
tokens-after-expr)))]

                 ;; ID '(' <Expression List> ')'
                 [(eq? (car next-token) 'lparen)
                  (let ((tokens-after-lparen (next-tokens tokens-after-id)))
                    (let-values (((expr-list tokens-after-expr-list)
(parse-expression-list tokens-after-lparen)))
                      (let ((tokens-after-rparen (match-token
tokens-after-expr-list 'rparen ")")))
                        (values (list 'CALL id-token expr-list)
tokens-after-rparen))))]
```

```
                 ;; Else, it's just an ID (which might be an error or
standalone expression)
                 [else
                  (values id-token tokens-after-id)]))
             ;; If no tokens after ID, it's just an ID
             (values id-token tokens-after-id)))]

      ;; Unknown statement type
      [else
       (error "Unknown statement type starting with" token)])))

;; Parse <ID List>
(define (parse-id-list tokens)
  (let-values (((id-token tokens-after-id) (match-token-type tokens 'id)))
    (let loop ((ids (list id-token))
               (tokens tokens-after-id))
      (if (not (tokens-empty? tokens))
          (let ((token (current-token tokens)))
            (if (eq? (car token) 'comma)
                (let ((tokens-after-comma (next-tokens tokens)))
                  (let-values (((next-id-token tokens-after-next-id)
(match-token-type tokens-after-comma 'id)))
                    (loop (append ids (list next-id-token))
tokens-after-next-id)))
                (values ids tokens)))
          (values ids tokens)))))

;; Parse <Expression List>
(define (parse-expression-list tokens)
  (let-values (((expr tokens-after-expr) (parse-expression tokens)))
    (let loop ((exprs (list expr))
               (tokens tokens-after-expr))
      (if (not (tokens-empty? tokens))
          (let ((token (current-token tokens)))
            (if (eq? (car token) 'comma)
                (let ((tokens-after-comma (next-tokens tokens)))
                  (let-values (((next-expr tokens-after-next-expr)
(parse-expression tokens-after-comma)))
                    (loop (append exprs (list next-expr))
tokens-after-next-expr)))
                (values exprs tokens)))
          (values exprs tokens)))))

;; Parse <Print List>
```

```
(define (parse-print-list tokens)
  (if (tokens-empty? tokens)
      (values '() tokens)
      (let-values (((expr tokens-after-expr) (parse-expression tokens)))
        (let ((tokens tokens-after-expr))
          (if (not (tokens-empty? tokens))
              (let ((token (current-token tokens)))
                (if (eq? (car token) 'semicolon)
                    (let ((tokens-after-semicolon (next-tokens tokens)))
                      (let-values (((next-print-list
tokens-after-next-print-list)
                                    (parse-print-list
tokens-after-semicolon)))
                        (values (cons expr next-print-list)
tokens-after-next-print-list)))
                    (values (list expr) tokens)))
              (values (list expr) tokens))))))

;; Operator precedence lists
(define compare-operators '("=" "<>" "><" ">" ">=" "<" "<="))
(define add-operators '("+" "-"))
(define mult-operators '("*" "/"))

;; Parse <Expression>
(define (parse-expression tokens)
  (let-values (((left-exp tokens-after-left) (parse-and-exp tokens)))
    (if (not (tokens-empty? tokens-after-left))
        (let ((token (current-token tokens-after-left)))
          (if (and (eq? (car token) 'keyword) (string-ci=? (cadr token) "or"))
              (let ((tokens-after-or (next-tokens tokens-after-left)))
                (let-values (((right-exp tokens-after-right) (parse-expression
tokens-after-or)))
                  (values (list 'OR left-exp right-exp) tokens-after-right)))
              (values left-exp tokens-after-left)))
        (values left-exp tokens-after-left))))

;; Parse <And Exp>
(define (parse-and-exp tokens)
  (let-values (((left-exp tokens-after-left) (parse-not-exp tokens)))
    (if (not (tokens-empty? tokens-after-left))
        (let ((token (current-token tokens-after-left)))
          (if (and (eq? (car token) 'keyword) (string-ci=? (cadr token)
"and"))
              (let ((tokens-after-and (next-tokens tokens-after-left)))
                (let-values (((right-exp tokens-after-right) (parse-and-exp
tokens-after-and)))
                  (values (list 'AND left-exp right-exp) tokens-after-right)))
              (values left-exp tokens-after-left)))
        (values left-exp tokens-after-left))))

;; Parse <Not Exp>
(define (parse-not-exp tokens)
  (let ((token (current-token tokens)))
    (if (and (eq? (car token) 'keyword) (string-ci=? (cadr token) "not"))
        (let ((tokens-after-not (next-tokens tokens)))
          (let-values (((expr tokens-after-expr) (parse-compare-exp
tokens-after-not)))
            (values (list 'NOT expr) tokens-after-expr)))
        (parse-compare-exp tokens))))

;; Parse <Compare Exp>
(define (parse-compare-exp tokens)
  (let-values (((left-exp tokens-after-left) (parse-add-exp tokens)))
    (if (not (tokens-empty? tokens-after-left))
        (let ((token (current-token tokens-after-left)))
          (if (and (eq? (car token) 'operator)
                   (member (cadr token) compare-operators))
              (let ((operator-token (cadr token))
                    (tokens-after-operator (next-tokens tokens-after-left)))
                (let-values (((right-exp tokens-after-right) (parse-add-exp
tokens-after-operator)))
                  (values (list 'COMPARE left-exp operator-token right-exp)
                          tokens-after-right)))
              (values left-exp tokens-after-left)))
        (values left-exp tokens-after-left))))

;; Parse <Add Exp>
(define (parse-add-exp tokens)
  (let-values (((left-exp tokens-after-left) (parse-mult-exp tokens)))
    (let loop ((left-exp left-exp)
               (tokens tokens-after-left))
      (if (not (tokens-empty? tokens))
          (let ((token (current-token tokens)))
            (if (member (car token) '(plus minus))
                (let ((operator (cadr token))
                      (tokens-after-operator (next-tokens tokens)))
                  (let-values (((right-exp tokens-after-right) (parse-mult-exp
tokens-after-operator)))
```

```
                    (loop (list 'ADD left-exp (operator ,operator) right-exp)
tokens-after-right)))
                (values left-exp tokens)))
          (values left-exp tokens)))))

;; Parse <Mult Exp>
(define (parse-mult-exp tokens)
  (let-values (((left-exp tokens-after-left) (parse-negate-exp tokens)))
    (let loop ((left-exp left-exp)
               (tokens tokens-after-left))
      (if (not (tokens-empty? tokens))
          (let ((token (current-token tokens)))
            (if (member (car token) '(times divide))
                (let ((operator (cadr token))
                      (tokens-after-operator (next-tokens tokens)))
                  (let-values (((right-exp tokens-after-right)
(parse-negate-exp tokens-after-operator)))
                    (loop (list 'MULT left-exp (operator ,operator) right-exp)
tokens-after-right)))
                (values left-exp tokens)))
          (values left-exp tokens)))))

;; Parse <Negate Exp>
(define (parse-negate-exp tokens)
  (let ((token (current-token tokens)))
    (if (eq? (car token) 'minus)
        (let ((tokens-after-minus (next-tokens tokens)))
          (let-values (((value tokens-after-value) (parse-negate-exp
tokens-after-minus)))
            (values (list 'NEGATE value) tokens-after-value)))
        (parse-value tokens))))

;; Parse <Value>
(define (parse-value tokens)
  (let ((token (current-token tokens)))
    (cond
      ;; '(' <Expression> ')'
      [(eq? (car token) 'lparen)
       (let ((tokens-after-lparen (next-tokens tokens)))
         (let-values (((expr tokens-after-expr) (parse-expression
tokens-after-lparen)))
           (let ((tokens-after-rparen (match-token tokens-after-expr 'rparen
")")))
             (values expr tokens-after-rparen))))]
      ;; ID '(' <Expression List> ')'
      [(and (eq? (car token) 'id))
       (let ((id-token token)
             (tokens-after-id (next-tokens tokens)))
         (if (not (tokens-empty? tokens-after-id))
             (let ((next-token (current-token tokens-after-id)))
               (if (eq? (car next-token) 'lparen)
                   (let ((tokens-after-lparen (next-tokens tokens-after-id)))
                     (let-values (((expr-list tokens-after-expr-list)
(parse-expression-list tokens-after-lparen)))
                       (let ((tokens-after-rparen (match-token
tokens-after-expr-list 'rparen ")")))
                         (values (list 'CALL id-token expr-list)
tokens-after-rparen))))
                   (values id-token tokens-after-id)))
             (values id-token tokens-after-id)))]

      ;; ID
      [(eq? (car token) 'id)
       (values token (next-tokens tokens))]

      ;; Constants
      [(member (car token) '(integer real string))
       (values token (next-tokens tokens))]

      ;; Error
      [else
       (error "Expected a value, got" token)])))

;; -------------------------------------------------------------------
;; Example Usage
;; -------------------------------------------------------------------

;; Sample input
(define input "REM example comments
x := 5
y := x + 1 : z := (y - 2) * 3
A := somefunction(z)
PRINT (z * 2)
PRINT \"A = \" ; A
END
DEF somefunction(a)
  WHILE a <= 0 DO
```

```
      a := a + 10
  ENDWHILE : REM just some fiddling around to test the language
  IF a > 5 THEN
    a := a + 2
    RETURN a * 3
  ENDIF
  RETURN a : REM else clause
ENDDEF")

;; Tokenize the input
(define tokens (tokenize input))

;; Uncomment the following line to see the tokens
(pretty-print tokens)

;; Parse the tokens starting from <Lines>
(define parse-tree (parse-lines tokens))

;; Display the parse tree
(displayln "Parse Tree:")
(pretty-print parse-tree)
```

CHATGPT:

```
#lang racket

;; ------------------------------------------------------------------
;; Parsing Functions
;; ------------------------------------------------------------------

;; Parse <Lines>
(define (parse-lines tokens)
  ;; (Function remains the same)
)

;; Parse <Statements>
(define (parse-statements tokens)
  ;; (Function remains the same)
)

;; Parse <Statement>
(define (parse-statement tokens)
  ;; (Function remains the same)
)
```

```
;; Parse <ID List>
(define (parse-id-list tokens)
  ;; (Function remains the same)
)

;; Parse <Expression List>
(define (parse-expression-list tokens)
  ;; (Function remains the same)
)

;; Parse <Print List>
(define (parse-print-list tokens)
  ;; (Function remains the same)
)

;; Parse <Expression>
(define (parse-expression tokens)
  ;; (Function remains the same)
)

;; Parse <And Exp>
(define (parse-and-exp tokens)
  ;; (Function remains the same)
)

;; Parse <Not Exp>
(define (parse-not-exp tokens)
  ;; (Function remains the same)
)

;; Parse <Compare Exp>
(define (parse-compare-exp tokens)
  ;; (Function remains the same)
)

;; Updated Parse <Add Exp>
(define (parse-add-exp tokens)
  (let-values (((left-exp tokens-after-left) (parse-mult-exp tokens)))
    (let loop ((left-exp left-exp)
               (tokens tokens-after-left))
      (if (tokens-empty? tokens)
          (values left-exp tokens)
          (let ((token (current-token tokens)))
            (if (or (eq? (car token) 'plus) (eq? (car token) 'minus))
                (let ((operator (cadr token))
                      (op-symbol (if (eq? (car token) 'plus) 'ADD 'SUB))
                      (tokens-after-operator (next-tokens tokens)))
                  (let-values (((right-exp tokens-after-right) (parse-mult-exp
tokens-after-operator)))
                    (loop (list op-symbol left-exp `(operator ,operator)
right-exp)
                          tokens-after-right)))
                (values left-exp tokens))))))

;; Updated Parse <Mult Exp>
(define (parse-mult-exp tokens)
  (let-values (((left-exp tokens-after-left) (parse-negate-exp tokens)))
    (let loop ((left-exp left-exp)
               (tokens tokens-after-left))
      (if (tokens-empty? tokens)
          (values left-exp tokens)
          (let ((token (current-token tokens)))
            (if (or (eq? (car token) 'times) (eq? (car token) 'divide))
                (let ((operator (cadr token))
                      (op-symbol (if (eq? (car token) 'times) 'MULT 'DIVIDE))
                      (tokens-after-operator (next-tokens tokens)))
                  (let-values (((right-exp tokens-after-right)
(parse-negate-exp tokens-after-operator)))
                    (loop (list op-symbol left-exp `(operator ,operator)
right-exp)
                          tokens-after-right)))
                (values left-exp tokens))))))

;; Parse <Negate Exp>
(define (parse-negate-exp tokens)
  (let ((token (current-token tokens)))
    (if (eq? (car token) 'minus)
        (let ((tokens-after-minus (next-tokens tokens)))
          (let-values (((value tokens-after-value) (parse-negate-exp
tokens-after-minus)))
            (values (list 'NEGATE value) tokens-after-value)))
        (parse-value tokens))))

;; Parse <Value>
(define (parse-value tokens)
  ;; (Function remains the same)
)
```

```
;; ---------------------------------------------------------------------
;; Example Usage
;; ---------------------------------------------------------------------

;; Sample input (as provided in your example)
(define input "REM example comments
x := 5
y := x + 1 : z := (y - 2) * 3
A := somefunction(z)
PRINT (z * 2)
PRINT \"A = \" ; A
END
DEF somefunction(a)
  WHILE a <= 0 DO
      a := a + 10
  ENDWHILE : REM just some fiddling around to test the language
  IF a > 5 THEN
    a := a + 2
    RETURN a * 3
  ENDIF
  RETURN a : REM else clause
ENDDEF")

;; Tokenize the input
(define tokens (tokenize input))

;; Parse the tokens starting from <Lines>
(define parse-tree (parse-lines tokens))

;; Display the parse tree
(displayln "Parse Tree:")
(pretty-print parse-tree)
```

Me:
Perfect. Thanks. How do we skip over comments/remarks?

CHATGPT:

To skip over comments in your parser (i.e., ignore comments so they don't appear in the parse tree), you'll need to adjust your code to ensure that comments are either not tokenized or are ignored during parsing.

You have two main options:

1. **Modify the Tokenizer to Exclude Comment Tokens:** Adjust the tokenizer so that it doesn't produce tokens for comments. This way, the parser will never encounter comments.
2. **Modify the Parser to Ignore Comment Tokens:** Keep the tokenizer as it is (producing comment tokens), but adjust the parser to skip over comment tokens and not include them in the parse tree.

ME:
Modify the tokenizer.

CHATGPT:

Full Tokenizer with Comment Skipping:

```
(define (tokenize input-string)
  (define (scan input tokens)
    (if (or (string-empty? input) (string-blank? input))
        (reverse tokens)
        (cond
          ;; Skip comments by not adding any token
          [(regexp-match #px"^(REM)[^\n]*" (string-upcase input))
           => (lambda (match)
                (let* ([matched (car match)]
                       [rest (substring input (string-length matched))])
                  ;; Continue scanning without adding a token
                  (scan rest tokens)))]
          ;; ... rest of the tokenizer code remains the same
          )))
  ;; Start scanning
  (scan input-string '()))
```

MID-PROJECT Note 2:

Note: From here I implemented the test code myself and my Scanner and Parser were complete.

- PROJECT END -