Program 2: Recursive descent, top-down parser

Hashim Abdulla

Due Date: 3/31/2025

**LLM Prompts**

(I first started generating code using Open AI’s GPT4o model, and DeepSeek but they both kept making syntax errors and couldn’t properly implement what I wanted no matter how many times I tried. So, I left those prompts and generations out and included only successful results from Claud’s 3.7 Sonnet model so this documentation wouldn’t be too long.)

Prompt 1: Initial Parsing Structure

“*I need to write a recursive-descent parser in Racket for a simple calculator language. The grammar is:*

*program -> {stmt\_list} $$*

*stmt\_list -> stmt stmt\_list | epsilon*

*stmt -> id = expr; | if (expr) stmt\_list endif; | read id; | write expr;*

*expr -> id etail | num etail*

*etail -> + expr | - expr | compare expr | epsilon*

*id -> [a-zA-Z]+*

*num -> numsign digit digit\**

*numsign -> + | - | epsilon*

*compare -> < | <= | > | >= | == | !=*

*help me get started with the basic structure of a recursive-descent parser in Racket”*

generated code:

1. #lang racket

2.

3. ;; Basic structure for a recursive descent parser

4.

5. ;; Parser state

6. (define current-token null)

7. (define input-port null)

8.

9. ;; Token scanning functions

10. (define (scan-token)

11. ;; To be implemented

12. '(EOF))

13.

14. (define (get-token)

15. (set! current-token (scan-token)))

16.

17. ;; Matching function

18. (define (match expected-token)

19. (if (equal? (car current-token) expected-token)

20. (begin

21. (get-token)

22. #t)

23. (error (format "Expected ~a, got ~a" expected-token (car current-token)))))

24.

25. ;; Parser functions for non-terminals

26. (define (parse-program)

27. (get-token)

28. ;; Implementation here

29. )

30.

31. (define (parse-stmt-list)

32. ;; Implementation here

33. )

34.

35. (define (parse-stmt)

36. ;; Implementation here

37. )

38.

39. (define (parse-expr)

40. ;; Implementation here

41. )

42.

43. (define (parse-etail)

44. ;; Implementation here

45. )

46.

47. ;; Main parsing function

48. (define (parse filename)

49. (set! input-port (open-input-file filename))

50. ;; Call the top-level parser function

51. (parse-program)

52. (close-input-port input-port)

53. "Accept")

54.

55. (module+ main

56. (parse "input.txt"))

57.

Prompt 2: Implementing Token Scanner

“*Now I need to implement the token scanner part. The tokens in my language include:*

*- Identifiers (letters)*

*- Numbers (digits with optional sign)*

*- Operators (+, -, <, <=, >, >=, ==, !=)*

*- Keywords (if, endif, read, write)*

*- Special characters (=, ;, (, ), $$)*

*How would I implement the scanner to recognize these tokens and track line numbers for error reporting?*”

generated code:

1. #lang racket

2.

3. ;; Recursive Descent Parser for a simple calculator language

4.

5. ;; Global variables for parser state

6. (define current-token null)

7. (define next-char #\space)

8. (define current-line 1)

9. (define current-pos 0)

10. (define input-port null)

11.

12. ;; Function to get the next character from the input

13. (define (get-char)

14. (set! next-char (read-char input-port))

15. (when (eof-object? next-char)

16. (set! next-char #\space))

17. (when (char=? next-char #\newline)

18. (set! current-line (add1 current-line))

19. (set! current-pos 0))

20. (set! current-pos (add1 current-pos)))

21.

22. ;; Function to skip whitespace

23. (define (skip-whitespace)

24. (when (and (not (eof-object? next-char)) (char-whitespace? next-char))

25. (get-char)

26. (skip-whitespace)))

27.

28. ;; Function to check if a character is a letter

29. (define (letter? c)

30. (and (char? c) (or (char<=? #\a c #\z) (char<=? #\A c #\Z))))

31.

32. ;; Function to check if a character is a digit

33. (define (digit? c)

34. (and (char? c) (char<=? #\0 c #\9)))

35.

36. ;; Function to scan the next token

37. (define (scan-token)

38. (skip-whitespace)

39. (cond

40. [(eof-object? next-char) '(EOF)]

41. [(letter? next-char) (scan-id)]

42. [(digit? next-char) (scan-num)]

43. [(char=? next-char #\+) (begin (get-char) '(PLUS))]

44. [(char=? next-char #\-) (begin (get-char) '(MINUS))]

45. [(char=? next-char #\() (begin (get-char) '(LPAREN))]

46. [(char=? next-char #\)) (begin (get-char) '(RPAREN))]

47. [(char=? next-char #\;) (begin (get-char) '(SEMICOLON))]

48. [(char=? next-char #\=) (begin

49. (get-char)

50. (if (char=? next-char #\=)

51. (begin (get-char) '(COMPARE ==))

52. '(ASSIGN)))]

53. [(char=? next-char #\<) (begin

54. (get-char)

55. (if (char=? next-char #\=)

56. (begin (get-char) '(COMPARE <=))

57. '(COMPARE <)))]

58. [(char=? next-char #\>) (begin

59. (get-char)

60. (if (char=? next-char #\=)

61. (begin (get-char) '(COMPARE >=))

62. '(COMPARE >)))]

63. [(char=? next-char #\!) (begin

64. (get-char)

65. (if (char=? next-char #\=)

66. (begin (get-char) '(COMPARE !=))

67. (error (format "Unexpected character '!'" current-line))))]

68. [else (error (format "Illegal character '~a'" next-char))]))

69.

70. ;; Function to scan an identifier

71. (define (scan-id)

72. (let loop ([id-str (string next-char)])

73. (get-char)

74. (if (letter? next-char)

75. (loop (string-append id-str (string next-char)))

76. (cond

77. [(string=? id-str "if") '(IF)]

78. [(string=? id-str "endif") '(ENDIF)]

79. [(string=? id-str "read") '(READ)]

80. [(string=? id-str "write") '(WRITE)]

81. [else `(ID ,id-str)]))))

82.

83. ;; Function to scan a number

84. (define (scan-num)

85. (let loop ([num-str (string next-char)])

86. (get-char)

87. (if (digit? next-char)

88. (loop (string-append num-str (string next-char)))

89. `(NUM ,(string->number num-str)))))

90.

91. ;; Function to get the next token

92. (define (get-token)

93. (set! current-token (scan-token)))

94.

95. ;; Placeholder for the rest of the parser

96. (define (parse filename)

97. (set! input-port (open-input-file filename))

98. (set! current-line 1)

99. (get-char) ; Initialize by getting the first character

100. ;; Rest of the parser to be implemented

101. (close-input-port input-port)

102. "Accept")

103.

Prompt 3: Implementing Basic Parser Functions

“*Now I need to implement the match function and the basic parser functions for each non-terminal in the grammar. I’d like the parser to build a parse tree that represents the structure of the program. Also, I need to handle the "$$" end marker properly.”*

generated code:

1. #lang racket

2.

3. ;; Recursive Descent Parser for a simple calculator language

4. ;; Grammar:

5. ;; program -> {stmt\_list} $$

6. ;; stmt\_list -> stmt stmt\_list | epsilon

7. ;; stmt -> id = expr; | if (expr) stmt\_list endif; | read id; | write expr;

8. ;; expr -> id etail | num etail

9. ;; etail -> + expr | - expr | compare expr | epsilon

10.

11. ;; Global variables for parser state

12. (define current-token null)

13. (define next-char #\space)

14. (define current-line 1)

15. (define current-pos 0)

16. (define input-port null)

17.

18. ;; Function to get the next character from the input

19. (define (get-char)

20. (set! next-char (read-char input-port))

21. (when (eof-object? next-char)

22. (set! next-char #\space))

23. (when (char=? next-char #\newline)

24. (set! current-line (add1 current-line))

25. (set! current-pos 0))

26. (set! current-pos (add1 current-pos)))

27.

28. ;; Function to skip whitespace

29. (define (skip-whitespace)

30. (when (and (not (eof-object? next-char)) (char-whitespace? next-char))

31. (get-char)

32. (skip-whitespace)))

33.

34. ;; Function to check if a character is a letter

35. (define (letter? c)

36. (and (char? c) (or (char<=? #\a c #\z) (char<=? #\A c #\Z))))

37.

38. ;; Function to check if a character is a digit

39. (define (digit? c)

40. (and (char? c) (char<=? #\0 c #\9)))

41.

42. ;; Function to scan the next token

43. (define (scan-token)

44. (skip-whitespace)

45. (cond

46. [(eof-object? next-char) '(EOF)]

47. [(letter? next-char) (scan-id)]

48. [(digit? next-char) (scan-num)]

49. [(char=? next-char #\+) (begin (get-char) '(PLUS))]

50. [(char=? next-char #\-) (begin (get-char) '(MINUS))]

51. [(char=? next-char #\() (begin (get-char) '(LPAREN))]

52. [(char=? next-char #\)) (begin (get-char) '(RPAREN))]

53. [(char=? next-char #\;) (begin (get-char) '(SEMICOLON))]

54. [(char=? next-char #\=) (begin

55. (get-char)

56. (if (char=? next-char #\=)

57. (begin (get-char) '(COMPARE ==))

58. '(ASSIGN)))]

59. [(char=? next-char #\<) (begin

60. (get-char)

61. (if (char=? next-char #\=)

62. (begin (get-char) '(COMPARE <=))

63. '(COMPARE <)))]

64. [(char=? next-char #\>) (begin

65. (get-char)

66. (if (char=? next-char #\=)

67. (begin (get-char) '(COMPARE >=))

68. '(COMPARE >)))]

69. [(char=? next-char #\!) (begin

70. (get-char)

71. (if (char=? next-char #\=)

72. (begin (get-char) '(COMPARE !=))

73. (error (format "Scan Error on line ~a: Unexpected character '!'" current-line))))]

74. [(char=? next-char #\$) (begin

75. (get-char)

76. (if (char=? next-char #\$)

77. (begin (get-char) '(END))

78. (error (format "Scan Error on line ~a: Unexpected character '$'" current-line))))]

79. [else (error (format "Scan Error on line ~a: Illegal character '~a'" current-line next-char))]))

80.

81. ;; Function to scan an identifier

82. (define (scan-id)

83. (let loop ([id-str (string next-char)])

84. (get-char)

85. (if (letter? next-char)

86. (loop (string-append id-str (string next-char)))

87. (cond

88. [(string=? id-str "if") '(IF)]

89. [(string=? id-str "endif") '(ENDIF)]

90. [(string=? id-str "read") '(READ)]

91. [(string=? id-str "write") '(WRITE)]

92. [else `(ID ,id-str)]))))

93.

94. ;; Function to scan a number

95. (define (scan-num)

96. (let loop ([num-str (string next-char)])

97. (get-char)

98. (if (digit? next-char)

99. (loop (string-append num-str (string next-char)))

100. `(NUM ,(string->number num-str)))))

101.

102. ;; Function to get the next token

103. (define (get-token)

104. (set! current-token (scan-token)))

105.

106. ;; Function to match the expected token

107. (define (match expected-token)

108. (if (equal? (car current-token) expected-token)

109. (let ([matched-token current-token])

110. (get-token)

111. matched-token)

112. (error (format "Parse Error on line ~a: Expected ~a, got ~a"

113. current-line

114. expected-token

115. (car current-token)))))

116.

117. ;; Parser functions for each non-terminal in the grammar

118.

119. ;; Program -> {stmt\_list} $$

120. (define (parse-program)

121. (get-token)

122. (let ([tree (parse-stmt-list)])

123. (match 'END)

124. (if (null? tree)

125. '(program)

126. `(program ,tree))))

127.

128. ;; stmt\_list -> stmt stmt\_list | epsilon

129. (define (parse-stmt-list)

130. (if (or (equal? (car current-token) 'ID)

131. (equal? (car current-token) 'IF)

132. (equal? (car current-token) 'READ)

133. (equal? (car current-token) 'WRITE))

134. (let ([stmt (parse-stmt)]

135. [rest-stmts (parse-stmt-list)])

136. (if (null? rest-stmts)

137. `(stmt-list ,stmt)

138. `(stmt-list ,stmt ,rest-stmts)))

139. '()))

140.

141. ;; stmt -> id = expr; | if (expr) stmt\_list endif; | read id; | write expr;

142. (define (parse-stmt)

143. (cond

144. [(equal? (car current-token) 'ID)

145. (let ([id (match 'ID)])

146. (match 'ASSIGN)

147. (let ([expr (parse-expr)])

148. (match 'SEMICOLON)

149. `(assign-stmt ,id ,expr)))]

150. [(equal? (car current-token) 'IF)

151. (match 'IF)

152. (match 'LPAREN)

153. (let ([expr (parse-expr)])

154. (match 'RPAREN)

155. (let ([stmt-list (parse-stmt-list)])

156. (match 'ENDIF)

157. (match 'SEMICOLON)

158. `(if-stmt ,expr ,stmt-list)))]

159. [(equal? (car current-token) 'READ)

160. (match 'READ)

161. (let ([id (match 'ID)])

162. (match 'SEMICOLON)

163. `(read-stmt ,id))]

164. [(equal? (car current-token) 'WRITE)

165. (match 'WRITE)

166. (let ([expr (parse-expr)])

167. (match 'SEMICOLON)

168. `(write-stmt ,expr))]))

169.

170. ;; Main parse function (to be completed)

171. (define (parse filename)

172. (set! input-port (open-input-file filename))

173. (set! current-line 1)

174. (set! current-pos 0)

175. (get-char) ; Initialize by getting the first character

176. ;; Add full error handling and parse tree construction

177. (close-input-port input-port)

178. "Accept")

179.

Prompt 4: Completing the Parser Implementation

“*complete the parser implementation by adding:*

*1. The remaining parser functions for expr and etail*

*2. Better error handling in the parse function to catch syntax errors*

*3. Proper handling of negative numbers*

*4. A way to return the parse tree as part of the "Accept" message*

*5. Testing with multiple input files file1.txt, file2.txt, file3.txt, and file4.txt*

*Implement these final components”*

generated code:

1. #lang racket

2.

3. ;; Recursive Descent Parser for a simple calculator language

4. ;; Grammar:

5. ;; program -> {stmt\_list} $$

6. ;; stmt\_list -> stmt stmt\_list | epsilon

7. ;; stmt -> id = expr; | if (expr) stmt\_list endif; | read id; | write expr;

8. ;; expr -> id etail | num etail

9. ;; etail -> + expr | - expr | compare expr | epsilon

10. ;; id -> [a-zA-Z]+

11. ;; num -> numsign digit digit\*

12. ;; numsign -> + | - | epsilon

13. ;; compare -> < | <= | > | >= | == | !=

14.

15. ;; Define the structure to store the current parsing state

16. (struct parser-state (input current-line current-pos))

17.

18. ;; Global variables for parser state

19. (define current-token null)

20. (define next-char #\space)

21. (define current-line 1)

22. (define current-pos 0)

23. (define input-port null)

24.

25. ;; Function to get the next character from the input

26. (define (get-char)

27. (set! next-char (read-char input-port))

28. (when (eof-object? next-char)

29. (set! next-char #\space))

30. (when (char=? next-char #\newline)

31. (set! current-line (add1 current-line))

32. (set! current-pos 0))

33. (set! current-pos (add1 current-pos)))

34.

35. ;; Function to skip whitespace

36. (define (skip-whitespace)

37. (when (and (not (eof-object? next-char)) (char-whitespace? next-char))

38. (get-char)

39. (skip-whitespace)))

40.

41. ;; Function to check if a character is a letter

42. (define (letter? c)

43. (and (char? c) (or (char<=? #\a c #\z) (char<=? #\A c #\Z))))

44.

45. ;; Function to check if a character is a digit

46. (define (digit? c)

47. (and (char? c) (char<=? #\0 c #\9)))

48.

49. ;; Function to scan the next token

50. (define (scan-token)

51. (skip-whitespace)

52. (cond

53. [(eof-object? next-char) '(EOF)]

54. [(letter? next-char) (scan-id)]

55. [(digit? next-char) (scan-num)]

56. [(char=? next-char #\+) (begin (get-char) '(PLUS))]

57. [(char=? next-char #\-) (begin (get-char) '(MINUS))]

58. [(char=? next-char #\() (begin (get-char) '(LPAREN))]

59. [(char=? next-char #\)) (begin (get-char) '(RPAREN))]

60. [(char=? next-char #\;) (begin (get-char) '(SEMICOLON))]

61. [(char=? next-char #\=) (begin

62. (get-char)

63. (if (char=? next-char #\=)

64. (begin (get-char) '(COMPARE ==))

65. '(ASSIGN)))]

66. [(char=? next-char #\<) (begin

67. (get-char)

68. (if (char=? next-char #\=)

69. (begin (get-char) '(COMPARE <=))

70. '(COMPARE <)))]

71. [(char=? next-char #\>) (begin

72. (get-char)

73. (if (char=? next-char #\=)

74. (begin (get-char) '(COMPARE >=))

75. '(COMPARE >)))]

76. [(char=? next-char #\!) (begin

77. (get-char)

78. (if (char=? next-char #\=)

79. (begin (get-char) '(COMPARE !=))

80. (error (format "Scan Error on line ~a: Unexpected character '!'" current-line))))]

81. [(char=? next-char #\$) (begin

82. (get-char)

83. (if (char=? next-char #\$)

84. (begin (get-char) '(END))

85. (error (format "Scan Error on line ~a: Unexpected character '$'" current-line))))]

86. [else (error (format "Scan Error on line ~a: Illegal character '~a'" current-line next-char))]))

87.

88. ;; Function to scan an identifier

89. (define (scan-id)

90. (let loop ([id-str (string next-char)])

91. (get-char)

92. (if (letter? next-char)

93. (loop (string-append id-str (string next-char)))

94. (cond

95. [(string=? id-str "if") '(IF)]

96. [(string=? id-str "endif") '(ENDIF)]

97. [(string=? id-str "read") '(READ)]

98. [(string=? id-str "write") '(WRITE)]

99. [else `(ID ,id-str)]))))

100.

101. ;; Function to scan a number

102. (define (scan-num)

103. (let loop ([num-str (string next-char)])

104. (get-char)

105. (if (digit? next-char)

106. (loop (string-append num-str (string next-char)))

107. `(NUM ,(string->number num-str)))))

108.

109. ;; Function to get the next token

110. (define (get-token)

111. (set! current-token (scan-token)))

112.

113. ;; Function to match the expected token

114. (define (match expected-token)

115. (if (equal? (car current-token) expected-token)

116. (let ([matched-token current-token])

117. (get-token)

118. matched-token)

119. (error (format "Parse Error on line ~a: Expected ~a, got ~a"

120. current-line

121. expected-token

122. (car current-token)))))

123.

124. ;; Parser functions for each non-terminal in the grammar

125.

126. ;; Program -> {stmt\_list} $$

127. (define (parse-program)

128. (get-token)

129. (let ([tree (parse-stmt-list)])

130. (match 'END)

131. (if (null? tree)

132. '(program)

133. `(program ,tree))))

134.

135. ;; stmt\_list -> stmt stmt\_list | epsilon

136. (define (parse-stmt-list)

137. (if (or (equal? (car current-token) 'ID)

138. (equal? (car current-token) 'IF)

139. (equal? (car current-token) 'READ)

140. (equal? (car current-token) 'WRITE))

141. (let ([stmt (parse-stmt)]

142. [rest-stmts (parse-stmt-list)])

143. (if (null? rest-stmts)

144. `(stmt-list ,stmt)

145. `(stmt-list ,stmt ,rest-stmts)))

146. '()))

147.

148. ;; stmt -> id = expr; | if (expr) stmt\_list endif; | read id; | write expr;

149. (define (parse-stmt)

150. (cond

151. [(equal? (car current-token) 'ID)

152. (let ([id (match 'ID)])

153. (match 'ASSIGN)

154. (let ([expr (parse-expr)])

155. (match 'SEMICOLON)

156. `(assign-stmt ,id ,expr)))]

157. [(equal? (car current-token) 'IF)

158. (match 'IF)

159. (match 'LPAREN)

160. (let ([expr (parse-expr)])

161. (match 'RPAREN)

162. (let ([stmt-list (parse-stmt-list)])

163. (match 'ENDIF)

164. (match 'SEMICOLON)

165. `(if-stmt ,expr ,stmt-list)))]

166. [(equal? (car current-token) 'READ)

167. (match 'READ)

168. (let ([id (match 'ID)])

169. (match 'SEMICOLON)

170. `(read-stmt ,id))]

171. [(equal? (car current-token) 'WRITE)

172. (match 'WRITE)

173. (let ([expr (parse-expr)])

174. (match 'SEMICOLON)

175. `(write-stmt ,expr))]))

176.

177. ;; expr -> id etail | num etail

178. (define (parse-expr)

179. (let ([term (cond

180. [(equal? (car current-token) 'ID) (match 'ID)]

181. [(equal? (car current-token) 'NUM) (match 'NUM)]

182. [(equal? (car current-token) 'MINUS)

183. (match 'MINUS)

184. (let ([num (match 'NUM)])

185. `(NUM ,(- (cadr num))))]

186. [else (error (format "Parse Error on line ~a: Expected ID or NUM, got ~a"

187. current-line

188. (car current-token)))])])

189. (let ([etail (parse-etail)])

190. (if (null? etail)

191. `(expr ,term)

192. `(expr ,term ,etail)))))

193.

194. ;; etail -> + expr | - expr | compare expr | epsilon

195. (define (parse-etail)

196. (cond

197. [(equal? (car current-token) 'COMPARE)

198. (let ([compare-op (match 'COMPARE)]

199. [expr (parse-expr)])

200. `(etail ,compare-op ,expr))]

201. [(equal? (car current-token) 'PLUS)

202. (match 'PLUS)

203. (let ([expr (parse-expr)])

204. `(etail + ,expr))]

205. [(equal? (car current-token) 'MINUS)

206. (match 'MINUS)

207. (let ([expr (parse-expr)])

208. `(etail - ,expr))]

209. [else '()]))

210.

211. ;; Main parse function

212. (define (parse filename)

213. (with-handlers ([exn:fail? (lambda (e) (exn-message e))])

214. (set! input-port (open-input-file filename))

215. (set! current-line 1)

216. (set! current-pos 0)

217. (get-char) ; Initialize by getting the first character

218. (let ([parse-tree (parse-program)])

219. (close-input-port input-port)

220. (format "Accept\n~a" parse-tree))))

221.

222. ;; Test the parser with sample files

223. (module+ main

224. (displayln "Testing file1.txt:")

225. (displayln (parse "file1.txt"))

226.

227. (displayln "\nTesting file2.txt:")

228. (displayln (parse "file2.txt"))

229.

230. (displayln "\nTesting file3.txt:")

231. (displayln (parse "file3.txt"))

232.

233. (displayln "\nTesting file4.txt:")

234. (displayln (parse "file4.txt")))

235.

Prompt 5: Adding Parser Comments and Polish

“*Okay, Just wanna make sure we add some good comprehensive comments to explain the various parts of the grammar and how the parser works. There are some comments, but can you enhance the comments and make sure the code is well-documented? Also, I want to make sure my negative number handling is correct in the expr function*.”

generated code:

1. #lang racket

2.

3. ;; Recursive Descent Parser for a simple calculator language

4. ;; Grammar:

5. ;; program -> {stmt\_list} $$

6. ;; stmt\_list -> stmt stmt\_list | epsilon

7. ;; stmt -> id = expr; | if (expr) stmt\_list endif; | read id; | write expr;

8. ;; expr -> id etail | num etail

9. ;; etail -> + expr | - expr | compare expr | epsilon

10. ;; id -> [a-zA-Z]+

11. ;; num -> numsign digit digit\*

12. ;; numsign -> + | - | epsilon

13. ;; compare -> < | <= | > | >= | == | !=

14.

15. ;; Define the structure to store the current parsing state

16. (struct parser-state (input current-line current-pos))

17.

18. ;; Global variables for parser state

19. (define current-token null)

20. (define next-char #\space)

21. (define current-line 1)

22. (define current-pos 0)

23. (define input-port null)

24.

25. ;; Function to get the next character from the input

26. (define (get-char)

27. (set! next-char (read-char input-port))

28. (when (eof-object? next-char)

29. (set! next-char #\space))

30. (when (char=? next-char #\newline)

31. (set! current-line (add1 current-line))

32. (set! current-pos 0))

33. (set! current-pos (add1 current-pos)))

34.

35. ;; Function to skip whitespace

36. (define (skip-whitespace)

37. (when (and (not (eof-object? next-char)) (char-whitespace? next-char))

38. (get-char)

39. (skip-whitespace)))

40.

41. ;; Function to check if a character is a letter

42. (define (letter? c)

43. (and (char? c) (or (char<=? #\a c #\z) (char<=? #\A c #\Z))))

44.

45. ;; Function to check if a character is a digit

46. (define (digit? c)

47. (and (char? c) (char<=? #\0 c #\9)))

48.

49. ;; Function to scan the next token

50. (define (scan-token)

51. (skip-whitespace)

52. (cond

53. [(eof-object? next-char) '(EOF)]

54. [(letter? next-char) (scan-id)]

55. [(digit? next-char) (scan-num)]

56. [(char=? next-char #\+) (begin (get-char) '(PLUS))]

57. [(char=? next-char #\-) (begin (get-char) '(MINUS))]

58. [(char=? next-char #\() (begin (get-char) '(LPAREN))]

59. [(char=? next-char #\)) (begin (get-char) '(RPAREN))]

60. [(char=? next-char #\;) (begin (get-char) '(SEMICOLON))]

61. [(char=? next-char #\=) (begin

62. (get-char)

63. (if (char=? next-char #\=)

64. (begin (get-char) '(COMPARE ==))

65. '(ASSIGN)))]

66. [(char=? next-char #\<) (begin

67. (get-char)

68. (if (char=? next-char #\=)

69. (begin (get-char) '(COMPARE <=))

70. '(COMPARE <)))]

71. [(char=? next-char #\>) (begin

72. (get-char)

73. (if (char=? next-char #\=)

74. (begin (get-char) '(COMPARE >=))

75. '(COMPARE >)))]

76. [(char=? next-char #\!) (begin

77. (get-char)

78. (if (char=? next-char #\=)

79. (begin (get-char) '(COMPARE !=))

80. (error (format "Scan Error on line ~a: Unexpected character '!'" current-line))))]

81. [(char=? next-char #\$) (begin

82. (get-char)

83. (if (char=? next-char #\$)

84. (begin (get-char) '(END))

85. (error (format "Scan Error on line ~a: Unexpected character '$'" current-line))))]

86. [else (error (format "Scan Error on line ~a: Illegal character '~a'" current-line next-char))]))

87.

88. ;; Function to scan an identifier

89. (define (scan-id)

90. (let loop ([id-str (string next-char)])

91. (get-char)

92. (if (letter? next-char)

93. (loop (string-append id-str (string next-char)))

94. (cond

95. [(string=? id-str "if") '(IF)]

96. [(string=? id-str "endif") '(ENDIF)]

97. [(string=? id-str "read") '(READ)]

98. [(string=? id-str "write") '(WRITE)]

99. [else `(ID ,id-str)]))))

100.

101. ;; Function to scan a number

102. (define (scan-num)

103. (let loop ([num-str (string next-char)])

104. (get-char)

105. (if (digit? next-char)

106. (loop (string-append num-str (string next-char)))

107. `(NUM ,(string->number num-str)))))

108.

109. ;; Function to get the next token

110. (define (get-token)

111. (set! current-token (scan-token)))

112.

113. ;; Function to match the expected token

114. (define (match expected-token)

115. (if (equal? (car current-token) expected-token)

116. (let ([matched-token current-token])

117. (get-token)

118. matched-token)

119. (error (format "Parse Error on line ~a: Expected ~a, got ~a"

120. current-line

121. expected-token

122. (car current-token)))))

123.

124. ;; Parser functions for each non-terminal in the grammar

125.

126. ;; Program -> {stmt\_list} $$

127. (define (parse-program)

128. (get-token)

129. (let ([tree (parse-stmt-list)])

130. (match 'END)

131. (if (null? tree)

132. '(program)

133. `(program ,tree))))

134.

135. ;; stmt\_list -> stmt stmt\_list | epsilon

136. (define (parse-stmt-list)

137. (if (or (equal? (car current-token) 'ID)

138. (equal? (car current-token) 'IF)

139. (equal? (car current-token) 'READ)

140. (equal? (car current-token) 'WRITE))

141. (let ([stmt (parse-stmt)]

142. [rest-stmts (parse-stmt-list)])

143. (if (null? rest-stmts)

144. `(stmt-list ,stmt)

145. `(stmt-list ,stmt ,rest-stmts)))

146. '()))

147.

148. ;; stmt -> id = expr; | if (expr) stmt\_list endif; | read id; | write expr;

149. (define (parse-stmt)

150. (cond

151. [(equal? (car current-token) 'ID)

152. (let ([id (match 'ID)])

153. (match 'ASSIGN)

154. (let ([expr (parse-expr)])

155. (match 'SEMICOLON)

156. `(assign-stmt ,id ,expr)))]

157. [(equal? (car current-token) 'IF)

158. (match 'IF)

159. (match 'LPAREN)

160. (let ([expr (parse-expr)])

161. (match 'RPAREN)

162. (let ([stmt-list (parse-stmt-list)])

163. (match 'ENDIF)

164. (match 'SEMICOLON)

165. `(if-stmt ,expr ,stmt-list)))]

166. [(equal? (car current-token) 'READ)

167. (match 'READ)

168. (let ([id (match 'ID)])

169. (match 'SEMICOLON)

170. `(read-stmt ,id))]

171. [(equal? (car current-token) 'WRITE)

172. (match 'WRITE)

173. (let ([expr (parse-expr)])

174. (match 'SEMICOLON)

175. `(write-stmt ,expr))]))

176.

177. ;; expr -> id etail | num etail

178. (define (parse-expr)

179. (let ([term (cond

180. [(equal? (car current-token) 'ID) (match 'ID)]

181. [(equal? (car current-token) 'NUM) (match 'NUM)]

182. [(equal? (car current-token) 'MINUS)

183. (match 'MINUS)

184. (let ([num (match 'NUM)])

185. `(NUM ,(- (cadr num))))]

186. [else (error (format "Parse Error on line ~a: Expected ID or NUM, got ~a"

187. current-line

188. (car current-token)))])])

189. (let ([etail (parse-etail)])

190. (if (null? etail)

191. `(expr ,term)

192. `(expr ,term ,etail)))))

193.

194. ;; etail -> + expr | - expr | compare expr | epsilon

195. (define (parse-etail)

196. (cond

197. [(equal? (car current-token) 'COMPARE)

198. (let ([compare-op (match 'COMPARE)]

199. [expr (parse-expr)])

200. `(etail ,compare-op ,expr))]

201. [(equal? (car current-token) 'PLUS)

202. (match 'PLUS)

203. (let ([expr (parse-expr)])

204. `(etail + ,expr))]

205. [(equal? (car current-token) 'MINUS)

206. (match 'MINUS)

207. (let ([expr (parse-expr)])

208. `(etail - ,expr))]

209. [else '()]))

210.

211. ;; Main parse function

212. (define (parse filename)

213. (with-handlers ([exn:fail? (lambda (e) (exn-message e))])

214. (set! input-port (open-input-file filename))

215. (set! current-line 1)

216. (set! current-pos 0)

217. (get-char) ; Initialize by getting the first character

218. (let ([parse-tree (parse-program)])

219. (close-input-port input-port)

220. (format "Accept\n~a" parse-tree))))

221.

It then took out the testing code at the end but I manually pasted it back in in the final program!