MODEL USED: DeepSeek R1

grammar.rkt

```
;program: LBRAC stmt-list RBRAC END
program: stmt-list END
stmt-list: stmt stmt-list | Ø
stmt: id EQUALS expr SEMICOLON
    | IF LPAREN expr RPAREN stmt-list ENDIF SEMICOLON
    | READ id SEMICOLON
    | WRITE expr SEMICOLON
expr: id etail
    | num etail
num: [NUMSIGN] DIGIT+
etail: PLUS expr
     | MINUS expr
     | COMPARE expr
     | Ø
id: LETTER+
```

I used the official documentation for translating the assignment's grammar to brag's syntax as it was pretty clear (and AI wasn't very helpful). As an important note, I originally had it to where the first production accounted for the "{" and the "}" found in the assignment's grammar's first production rule: `program -> {stmt_list} $$`, but changed it later to reflect the parser input files which didn't have them (it's now commented out as seen above). (**NOTE TO GRADER**: if the parser input files given to you to test this program against have curly brackets, comment out line 20 and uncomment line 19 in grammar.rkt and uncomment lines 40 and 41 in tokenizer.rkt; this should fix the program to work with the correct grammar)

tokenizer.rkt

```
32  ;; TOKENIZER
33
34  (define (tokenize ip)
35    (port-count-lines! ip)
36    (define my-lexer
37      (lexer-src-pos
38
39       ;; misc. terminals
40       ; ["{" (token 'LBRAC lexeme)]
41       ; ["}" (token 'RBRAC lexeme)]
42       ["if" (token 'IF lexeme)]
43       ["(" (token 'LPAREN lexeme)]
44       [")" (token 'RPAREN lexeme)]
45       ["endif" (token 'ENDIF lexeme)]
46       ["read" (token 'READ lexeme)]
47       ["write" (token 'WRITE lexeme)]
48       [";" (token 'SEMICOLON lexeme)]
49       ["=" (token 'EQUALS lexeme)]
50
51       ;; comparison symbols
52       [(union "<=" ">=" "==" "!=" "<" ">") (token 'COMPARE lexeme)]
53
54       ;; letter
55       [(union (char-range #\a #\z) (char-range #\A #\Z))
56        (token 'LETTER lexeme)]
57
58       ;; numsign / plus / minus
59       ;; (from DeepSeek) -> solves my ambiguity problem
60       ;; by using peek-char for lookahead that won't modify the
61       ;; ip
62       [(union "+" "-")
63        (let ([next-char (peek-char ip)])
64          (if (and next-char (char-numeric? next-char))
65              ;; if followed by a digit, treat as NUMSIGN (part of a number)
66              (token 'NUMSIGN lexeme)
67              ;; otherwise, treat as PLUS or MINUS (arithmetic operation)
68              (token (if (equal? lexeme "+") 'PLUS 'MINUS) lexeme)))]
69
70       ;; digit
71       [numeric (token 'DIGIT lexeme)]
72
73       ;; whitespace
74       [whitespace (token 'WHITESPACE lexeme #:skip? #t)]
75
76       ;; end of program
77       ["$$" (token 'END lexeme)]
78
79       ;; eof
80       [(eof) (token 'EOF lexeme)]))
81
82    ;; Return a function that generates tokens
83    (define (next-token) (my-lexer ip))
84    next-token)
85
```

For my tokenizer, I used the br-parser-tools/lex library—a library that provides useful functions for pattern matching and creating tokenizers. For this, I did use a little AI generated code: the pattern matching for “+” and “-” found in the code block starting at line 62 was generated by DeepSeek R1. I originally mistakenly had it to where any “+” or “-” read in would be only lexed into “PLUS” or “MINUS” tokens respectively. This problem arose as I didn’t implement any means to disambiguate between lexing “+/-” into those tokens and “NUMSIGN” tokens (as they are both identified by +/-). The solution to this problem was to use a lookahead character; if the next character was numeric, then the token produced would be a NUMSIGN; otherwise, it would produce either a PLUS or a MINUS depending on the character read in.

user-interface.rkt

Underlying Parse Function w/ Graceful Exception Handling

```
(define (safe-parse parser-func tokenizer-func input-port)
  ;; buffer for looking up the lines at which errors occur
  (define buffer (port->lines (peeking-input-port input-port)))
  (with-handlers

      ;; catches all errors with the code executed
      ;; in the second body of the with-handlers function
      ;; -----
      ;; if error is found, prints out the location in
      ;; the input file where the error occured
      {[exn:fail? (lambda (e)
                    (printf "Error: ~a~n" (modify-exn-message (exn-message e) buffer)))
                  ]}

      ;; code to be executed with error handling
      ;; -----
      ;; tokenizes and parses input and produces and prints syntax tree
      ;; along with "ACCEPT" if successful
      {begin (printf "Parse Tree~n-----~n~a" (syntax->datum (parser-func (tokenizer-func input-port))))
             (printf "~n-----~nACCEPT")}
  ))
```

This is a higher order wrapper function that takes a parser function, a tokenizer function, and an input-port, which executes the former two on the latter to produce a parse tree while gracefully handling any parsing/scanning errors along the way. All of the code here was written using only the documentation and my knowledge of the language. I did, however, learn about the “with-handlers” function from AI when I asked for ideas on how to gracefully handle errors (this is further elaborated in the AI Log document). The buffer variable was later tacked on to satisfy the requirement of printing out the line at which errors occur; the default error message produced by brag parsers do provide the line number of the error, but only for parsing errors. Scanning errors, on the other hand (such as the one found when parsing file4.txt), don’t get that luxury, so I had to manually implement that, hence the use of the buffer for storing the data from the input-port.

Error Message Manipulation Function

```
(define (modify-exn-message exn-message buffer)
  (define split-exn (string-split exn-message " "))
  (define is-scanner-error (equal? "lexer:" (first split-exn)))

  ;; helper function for looking up
  ;; line number of error
  (define (get-line-number buffer invalid-token)
    (for/fold
      ([current-line-number 1]
       [found #f]
       #:result current-line-number)
      ([current-line buffer])
      (if (or (equal? found #t)
              (regexp-match? (format "~a" invalid-token) current-line))
          (values current-line-number #t)
          (values (+ current-line-number 1) #f))
    )
  )

  (if is-scanner-error
    [let*
      ([raw-invalid-token (ninth split-exn)]
       [invalid-token (first (string-split raw-invalid-token "\""))]
       [line-number (get-line-number buffer invalid-token)])
      (format "Scaning Error at Line ~a; Invalid Token \"~a\"" line-number invalid-token)]
    [let*
      ([raw-line-number (string-split (list-ref split-exn 10) "[line=")]
       [string-line-number (first raw-line-number)]
       [string-line-number-no-comma (first (string-split string-line-number ","))]
       [line-number (string->number string-line-number-no-comma)]
       [adjusted-line-number (- line-number 1)])
      (format "Parsing Error at/around Line ~a" adjusted-line-number)]
  )
)
```

This function parses through the default error messages that brag parsers output and transforms them into error messages that satisfy the criteria of the assignment. Like the last function, I wrote this function using my knowledge of the language (I can't imagine how annoying it would be to get a sensible implementation from an AI for this, so I didn't even try)

Bindings and Partial Applications

```
68  ;; create bindings for the parser and tokenizer/lexer
69  (define my-parser (make-rule-parser program))
70  (define my-tokenizer tokenize)
71
72
73  ;; partial application of safe-parse for use in the execution loop
74  (define parse (lambda (input-port) (safe-parse my-parser my-tokenizer input-port)))
75
76  ;; partial application of safe-parse called "parse" as per the assignment requirements
77  ;; for use in the DrRacket REPL
78  (define repl-parse (lambda (file-path)
79                       (safe-parse my-parser my-tokenizer
80                                   (open-input-file (format "./parser-input/~a" file-path)))
81                     )
82  )
```

This section of the code defines some parser functions by using partial application of the aforementioned safe-parse function. No AI was used for this code.

Program Execution Loop

```
85  ;; in:  takes a function; asks user if they want to continue
86  ;; out: if user does, calls function
87  (define (continue? func)
88    (display "\n---\nContinue? Y to Continue, Anything Else to Exit\n   >>>")
89    (define input (string-trim (read-line)))
90
91    (if [or (equal? input "Y")
92            (equal? input "y")]
93        (func)
94        (display "---\n"))
95    )
96
97
98  ;; in:  no args
99  ;; out: returns a valid string file name from user input
00  (define (get-file-path)
01    (display "Enter Parser Input File Name\n   >>>")
02    (define input (string-trim (read-line)))
03
04    ;; Construct the full path to the file in the parser-input directory
05    (define path (format "./parser-input/~a" input))
06
07    ;; Check if the file exists
08    (if (file-exists? path)
09        path
10        (begin
11          (printf "Error: File Doesn't Exist: ~a~n---~n" path)
12          (get-file-path))))
13
14
15  ;; in:  no args
16  ;; out: main program execution loop
17  (define (execution-loop)
18      (define file-path (get-file-path))
19      (define input-port (open-input-file file-path))
20      (begin (parse input-port)
21             (continue? execution-loop)))
```

This section contains code that is mostly recycled from the first program with a few changes here and there. I originally tried to use AI for something fancy with the get-file-path function using the racket/path library, but I couldn't get it to work, so I changed it back to something more conventional.