1. **create the code in racket for a \*recursive descent, top-down \*parser, the grammar for the parser is program -> {stmt\_list} $$ stmt\_list -> stmt stmt\_list stmt\_list -> epsilon stmt -> id = expr; | if (expr) stmt\_list endif; | read id; | write expr; expr -> id etail | num etail etail -> + expr | - expr | epsilon id -> [a-zA-Z]+ num -> numsign digit digit\* numsign -> + | - | epsilon - ChatGPT**

#lang racket

(define (parse-program tokens)

(define (match expected tokens)

(if (and (not (null? tokens)) (equal? (car tokens) expected))

(cdr tokens)

(error "Syntax error: Expected" expected)))

(define (parse-stmt-list tokens)

(cond [(null? tokens) tokens]

[(equal? (car tokens) '$$) tokens]

[else (parse-stmt-list (parse-stmt tokens))]))

(define (parse-stmt tokens)

(cond [(and (pair? tokens) (symbol? (car tokens)) (equal? (cadr tokens) '=))

(parse-etoken (parse-expr (match '= tokens)))]

[(equal? (car tokens) 'if)

(let ([rest (parse-expr (match 'if tokens))])

(let ([rest2 (parse-stmt-list (match ') rest))])

(parse-etoken (match 'endif rest2))))]

[(equal? (car tokens) 'read)

(parse-etoken (match 'read tokens))]

[(equal? (car tokens) 'write)

(parse-etoken (parse-expr (match 'write tokens)))]

[else (error "Syntax error in statement" (car tokens))]))

(define (parse-expr tokens)

(let ([rest (parse-etail (cdr tokens))])

rest))

(define (parse-etail tokens)

(cond [(null? tokens) tokens]

[(or (equal? (car tokens) '+) (equal? (car tokens) '-))

(parse-expr (cdr tokens))]

[else tokens]))

(define (parse-etoken tokens)

(if (equal? (car tokens) ';)

(cdr tokens)

(error "Syntax error: Expected ;")))

(parse-stmt-list tokens))

; Example usage:

(define tokens '(x = 5 ; y = x + 3 ; if ( x - 2 ) y = x ; endif ; $$))

(parse-program tokens)

1. **I have 4 input files that need to be parsed so make it so that we have to parse these 4 input files and we get a parse tree, as an output and include messages as the parser is parsing the txt files. I want the output to be printed to a txt file, labeled output 1 to 4.**

#lang racket

(define (parse-program tokens output-file)

(define (write-log message)

(displayln message (current-output-port)))

(define (match expected tokens)

(if (and (not (null? tokens)) (equal? (car tokens) expected))

(cdr tokens)

(error "Syntax error: Expected" expected)))

(define (parse-stmt-list tokens)

(cond [(null? tokens) tokens]

[(equal? (car tokens) '$$) tokens]

[else (parse-stmt-list (parse-stmt tokens))]))

(define (parse-stmt tokens)

(write-log (format "Parsing statement: ~a" (car tokens)))

(cond [(and (pair? tokens) (symbol? (car tokens)) (equal? (cadr tokens) '=))

(parse-etoken (parse-expr (match '= tokens)))]

[(equal? (car tokens) 'if)

(let ([rest (parse-expr (match 'if tokens))])

(let ([rest2 (parse-stmt-list (match ') rest))])

(parse-etoken (match 'endif rest2))))]

[(equal? (car tokens) 'read)

(parse-etoken (match 'read tokens))]

[(equal? (car tokens) 'write)

(parse-etoken (parse-expr (match 'write tokens)))]

[else (error "Syntax error in statement" (car tokens))]))

(define (parse-expr tokens)

(write-log (format "Parsing expression: ~a" (car tokens)))

(let ([rest (parse-etail (cdr tokens))])

rest))

(define (parse-etail tokens)

(cond [(null? tokens) tokens]

[(or (equal? (car tokens) '+) (equal? (car tokens) '-))

(parse-expr (cdr tokens))]

[else tokens]))

(define (parse-etoken tokens)

(if (equal? (car tokens) ';)

(cdr tokens)

(error "Syntax error: Expected ;")))

(write-log "Parsing program...")

(parse-stmt-list tokens)

(write-log "Parsing complete."))

(define (read-input-file filename)

(with-input-from-file filename

(lambda () (map string->symbol (string-split (read-line) " "))))))

(define (parse-and-output input-file output-file)

(let ([tokens (read-input-file input-file)])

(with-output-to-file output-file

(lambda () (parse-program tokens output-file))

#:mode 'text)))

; Process the 4 input files

(define input-files '("input1.txt" "input2.txt" "input3.txt" "input4.txt"))

(define output-files '("output1.txt" "output2.txt" "output3.txt" "output4.txt"))

(map parse-and-output input-files output-files)

1. **([rest2 (parse-stmt-list (match ') rest))]) p2.rkt:23:51: read-syntax: unexpected `)`#(860 1)**

(let ([rest2 (parse-stmt-list (match ')' rest))]) - was mising the ‘ after the )

1. **Got the same error again, on the same exact line**

(let ([rest2 (parse-stmt-list (match '() rest))])) ; Corrected

1. **read-syntax: missing ] to close preceding [, found instead ) Interactions disabled.**

(let ([rest2 (parse-stmt-list (match 'endif rest))])

(parse-etoken rest2))

1. **begin (possibly implicit): the last form is not an expression in: (define (read-input-file filename) (with-input-from-file filename (lambda () (map string->symbol (string-split (read-line) " "))))) >**

(define (read-input-file filename)

(with-input-from-file filename

(lambda ()

(let ([line (read-line)])

(if (eof-object? line)

'()

(map string->symbol (string-split line " "))))))))

1. **begin (possibly implicit): the last form is not an expression in: (define (write-log message) (displayln message (current-output-port))) >**

(define (write-log message)

(displayln message))

1. **The code given by chatGpt was filled with syntax errors please fix them - Claude**

#lang racket

(define (parse-program tokens)

(define (write-log message)

(displayln message (current-output-port)))

(define (match expected tokens)

(if (and (not (null? tokens)) (equal? (car tokens) expected))

(cdr tokens)

(error (format "Syntax error: Expected ~a" expected))))

(define (parse-stmt-list tokens)

(cond [(null? tokens) tokens]

[(equal? (car tokens) '$$) tokens]

[else (parse-stmt-list (parse-stmt tokens))]))

(define (parse-stmt tokens)

(write-log (format "Parsing statement: ~a" (car tokens)))

(cond [(and (pair? tokens) (symbol? (car tokens)) (equal? (cadr tokens) '=))

(let ([var-name (car tokens)])

(parse-etoken (parse-expr (cddr tokens))))] ; Skip variable name and = sign

[(equal? (car tokens) 'if)

(let\* ([rest (parse-expr (cdr tokens))] ; Skip 'if'

[rest2 (parse-stmt-list (match 'then rest))])

(parse-etoken (match 'endif rest2)))]

[(equal? (car tokens) 'read)

(parse-etoken (cdr tokens))] ; Skip 'read'

[(equal? (car tokens) 'write)

(parse-etoken (parse-expr (cdr tokens)))] ; Skip 'write'

[else (error (format "Syntax error in statement: ~a" (car tokens)))]))

(define (parse-expr tokens)

(write-log (format "Parsing expression: ~a" (car tokens)))

(let ([first-token (car tokens)])

(cond

[(number? first-token) (parse-etail (cdr tokens))]

[(symbol? first-token) (parse-etail (cdr tokens))]

[(equal? first-token '() (parse-expr (cdr tokens)))]

[else (error (format "Invalid expression: ~a" first-token))])))

(define (parse-etail tokens)

(cond [(null? tokens) tokens]

[(or (equal? (car tokens) '+) (equal? (car tokens) '-))

(parse-etail (parse-expr (cdr tokens)))]

[else tokens]))

(define (parse-etoken tokens)

(if (and (not (null? tokens)) (equal? (car tokens) ';))

(cdr tokens)

(error "Syntax error: Expected ;")))

(write-log "Parsing program...")

(let ([result (parse-stmt-list tokens)])

(if (and (not (null? result)) (equal? (car result) '$$))

(write-log "Parsing complete.")

(error "Syntax error: Expected $$ at end of program")))

#t)

(define (read-input-file filename)

(with-input-from-file filename

(lambda ()

(map (lambda (token)

(cond

[(regexp-match #rx"^[0-9]+$" token) (string->number token)]

[else (string->symbol token)]))

(string-split (read-line) " ")))))

(define (parse-and-output input-file output-file)

(let ([tokens (read-input-file input-file)])

(with-output-to-file output-file

(lambda ()

(displayln (format "Processing file: ~a" input-file))

(parse-program tokens))

#:mode 'text

#:exists 'replace)))

; Process the 4 input files

(define input-files '("input1.txt" "input2.txt" "input3.txt" "input4.txt"))

(define output-files '("output1.txt" "output2.txt" "output3.txt" "output4.txt"))

(for-each (lambda (in-file out-file)

(parse-and-output in-file out-file))

input-files

output-files)

1. **create the code in racket for a \*recursive descent, top-down \*parser, it should work with all 4 files, this is the updated grammar for the parser program -> {stmt\_list} $$**

stmt\_list -> stmt stmt\_list

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 -> < | <= | > | >= | == | !=

#lang racket

;; Lexical analyzer

(define (tokenize input-string)

(define tokens '())

(define current-line 1)

(define line-positions '())

(define (add-token type value)

(set! tokens (append tokens (list (list type value current-line)))))

(define (add-line-position pos)

(set! line-positions (append line-positions (list (cons current-line pos)))))

(let loop ((i 0))

(when (< i (string-length input-string))

(let ((c (string-ref input-string i)))

(cond

;; New line

[(char=? c #\newline)

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

(add-line-position (add1 i))

(loop (add1 i))]

;; Whitespace

[(char-whitespace? c)

(loop (add1 i))]

;; Identifiers: letters

[(char-alphabetic? c)

(let id-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-alphabetic? (string-ref input-string j)))

(id-loop (add1 j))

(begin

(add-token 'ID (substring input-string i j))

(loop j))))]

;; Numbers: digits

[(char-numeric? c)

(let num-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-numeric? (string-ref input-string j)))

(num-loop (add1 j))

(begin

(add-token 'NUM (substring input-string i j))

(loop j))))]

;; Signs and operators

[(char=? c #\+) (add-token 'PLUS "+") (loop (add1 i))]

[(char=? c #\-) (add-token 'MINUS "-") (loop (add1 i))]

[(char=? c #\=)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'EQUALS "==")

(loop (+ i 2)))

(begin

(add-token 'ASSIGN "=")

(loop (add1 i))))]

[(char=? c #\<)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'LESSEQ "<=")

(loop (+ i 2)))

(begin

(add-token 'LESS "<")

(loop (add1 i))))]

[(char=? c #\>)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'GREATEQ ">=")

(loop (+ i 2)))

(begin

(add-token 'GREATER ">")

(loop (add1 i))))]

[(char=? c #\!)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'NOTEQ "!=")

(loop (+ i 2)))

(begin

(add-token 'ERROR "!")

(loop (add1 i))))]

;; Special characters

[(char=? c #\;) (add-token 'SEMICOLON ";") (loop (add1 i))]

[(char=? c #\() (add-token 'LPAREN "(") (loop (add1 i))]

[(char=? c #\)) (add-token 'RPAREN ")") (loop (add1 i))]

[(char=? c #\$)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\$))

(begin

(add-token 'ENDMARKER "$$")

(loop (+ i 2)))

(begin

(add-token 'ERROR "$")

(loop (add1 i))))]

;; Invalid character

[else

(add-token 'ERROR (string c))

(loop (add1 i))]))))

(values tokens line-positions))

;; Parser

(define (parse filename)

(define input (file->string filename))

(define-values (tokens line-positions) (tokenize input))

(define current-token-index 0)

(define parse-error #f)

;; Look at current token without consuming it

(define (peek-token)

(if (< current-token-index (length tokens))

(list-ref tokens current-token-index)

(list 'EOF "" 0)))

;; Consume and return the current token

(define (consume-token)

(let ((token (peek-token)))

(set! current-token-index (add1 current-token-index))

token))

;; Match a specific token type

(define (match-token expected-type)

(let ((token (peek-token)))

(if (eq? (first token) expected-type)

(consume-token)

(begin

(set! parse-error (format "Syntax error at line ~a: expected ~a but got ~a"

(third token)

expected-type

(first token)))

#f))))

;; Get the line content for error reporting

(define (get-line-content line-number)

(let loop ((line-pos-pairs line-positions)

(start 0)

(end (string-length input)))

(cond

[(null? line-pos-pairs) (substring input start end)]

[(= (caar line-pos-pairs) line-number)

(substring input start (cdar line-pos-pairs))]

[else (loop (cdr line-pos-pairs) (cdar line-pos-pairs) end)])))

;; Grammar implementation - recursive descent functions

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

(define (parse-program)

(let ((result (parse-stmt-list)))

(if (and result (match-token 'ENDMARKER))

(list 'PROGRAM result)

#f)))

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

(define (parse-stmt-list)

(let ((token (peek-token)))

(if (or (eq? (first token) 'ID)

(eq? (first token) 'ENDMARKER)

(eq? (first token) 'EOF)

(eq? (first token) 'ENDIF))

(let ((stmt (parse-stmt)))

(if stmt

(let ((stmt-list (parse-stmt-list)))

(if stmt-list

(list 'STMT\_LIST stmt stmt-list)

(list 'STMT\_LIST stmt)))

'(STMT\_LIST))) ; epsilon case

(if (or (eq? (first token) 'IF)

(eq? (first token) 'READ)

(eq? (first token) 'WRITE))

(let ((stmt (parse-stmt)))

(if stmt

(let ((stmt-list (parse-stmt-list)))

(if stmt-list

(list 'STMT\_LIST stmt stmt-list)

(list 'STMT\_LIST stmt)))

'(STMT\_LIST))) ; epsilon case

'(STMT\_LIST))))) ; epsilon case

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

(define (parse-stmt)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if (and id

(match-token 'ASSIGN)

(let ((expr (parse-expr)))

(if expr

(if (match-token 'SEMICOLON)

(list 'ASSIGN\_STMT (second id) expr)

#f)

#f)))

(list 'ASSIGN\_STMT (second id) (third expr))

#f))]

[(eq? (first token) 'IF)

(consume-token) ; consume IF

(if (match-token 'LPAREN)

(let ((expr (parse-expr)))

(if (and expr (match-token 'RPAREN))

(let ((stmt-list (parse-stmt-list)))

(if (and stmt-list (match-token 'ENDIF) (match-token 'SEMICOLON))

(list 'IF\_STMT expr stmt-list)

#f))

#f))

#f)]

[(eq? (first token) 'READ)

(consume-token) ; consume READ

(let ((id (match-token 'ID)))

(if (and id (match-token 'SEMICOLON))

(list 'READ\_STMT (second id))

#f))]

[(eq? (first token) 'WRITE)

(consume-token) ; consume WRITE

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(list 'WRITE\_STMT expr)

#f))]

[else #f])))

;; expr -> id etail | num etail

(define (parse-expr)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if id

(let ((etail (parse-etail)))

(if etail

(list 'EXPR (list 'ID (second id)) etail)

(list 'EXPR (list 'ID (second id)))))

#f))]

[(or (eq? (first token) 'NUM)

(eq? (first token) 'PLUS)

(eq? (first token) 'MINUS))

(let ((num (parse-num)))

(if num

(let ((etail (parse-etail)))

(if etail

(list 'EXPR num etail)

(list 'EXPR num)))

#f))]

[else #f])))

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

(define (parse-etail)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((expr (parse-expr)))

(if expr

(list 'ETAIL '+ expr)

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((expr (parse-expr)))

(if expr

(list 'ETAIL '- expr)

#f))]

[(or (eq? (first token) 'LESS)

(eq? (first token) 'LESSEQ)

(eq? (first token) 'GREATER)

(eq? (first token) 'GREATEQ)

(eq? (first token) 'EQUALS)

(eq? (first token) 'NOTEQ))

(let ((compare (parse-compare)))

(if compare

(let ((expr (parse-expr)))

(if expr

(list 'ETAIL compare expr)

#f))

#f))]

[else '()]))) ; epsilon case

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

;; Handled by tokenizer

;; num -> numsign digit digit\*

(define (parse-num)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "+" (second num))))

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "-" (second num))))

#f))]

[(eq? (first token) 'NUM)

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (second num)))

#f))]

[else #f])))

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

(define (parse-compare)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'LESS)

(consume-token)

'<]

[(eq? (first token) 'LESSEQ)

(consume-token)

'<=]

[(eq? (first token) 'GREATER)

(consume-token)

'>]

[(eq? (first token) 'GREATEQ)

(consume-token)

'>=]

[(eq? (first token) 'EQUALS)

(consume-token)

'==]

[(eq? (first token) 'NOTEQ)

(consume-token)

'!=]

[else #f])))

;; Start parsing

(let ((parse-tree (parse-program)))

(if parse-error

(let\* ((error-token (list-ref tokens (sub1 current-token-index)))

(line-number (third error-token))

(line-content (get-line-content line-number)))

(string-append parse-error "\nLine " (number->string line-number) ": " line-content))

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

;; Main function - exports the parse function for use

(provide parse)

1. **3-unsaved-editor:201:53: expr: unbound identifier**

**in: expr #(7311 4)**

#lang racket

;; Lexical analyzer

(define (tokenize input-string)

(define tokens '())

(define current-line 1)

(define line-positions '())

(define (add-token type value)

(set! tokens (append tokens (list (list type value current-line)))))

(define (add-line-position pos)

(set! line-positions (append line-positions (list (cons current-line pos)))))

(let loop ((i 0))

(when (< i (string-length input-string))

(let ((c (string-ref input-string i)))

(cond

;; New line

[(char=? c #\newline)

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

(add-line-position (add1 i))

(loop (add1 i))]

;; Whitespace

[(char-whitespace? c)

(loop (add1 i))]

;; Identifiers: letters

[(char-alphabetic? c)

(let id-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-alphabetic? (string-ref input-string j)))

(id-loop (add1 j))

(begin

(add-token 'ID (substring input-string i j))

(loop j))))]

;; Numbers: digits

[(char-numeric? c)

(let num-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-numeric? (string-ref input-string j)))

(num-loop (add1 j))

(begin

(add-token 'NUM (substring input-string i j))

(loop j))))]

;; Special keywords

[(char=? c #\i)

(if (and (< (+ i 1) (string-length input-string))

(char=? (string-ref input-string (+ i 1)) #\f))

(begin

(add-token 'IF "if")

(loop (+ i 2)))

(let id-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-alphabetic? (string-ref input-string j)))

(id-loop (add1 j))

(begin

(add-token 'ID (substring input-string i j))

(loop j)))))]

[(and (>= (+ i 5) (string-length input-string))

(string=? (substring input-string i (+ i 5)) "endif"))

(add-token 'ENDIF "endif")

(loop (+ i 5))]

[(and (>= (+ i 4) (string-length input-string))

(string=? (substring input-string i (+ i 4)) "read"))

(add-token 'READ "read")

(loop (+ i 4))]

[(and (>= (+ i 5) (string-length input-string))

(string=? (substring input-string i (+ i 5)) "write"))

(add-token 'WRITE "write")

(loop (+ i 5))]

;; Signs and operators

[(char=? c #\+) (add-token 'PLUS "+") (loop (add1 i))]

[(char=? c #\-) (add-token 'MINUS "-") (loop (add1 i))]

[(char=? c #\=)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'EQUALS "==")

(loop (+ i 2)))

(begin

(add-token 'ASSIGN "=")

(loop (add1 i))))]

[(char=? c #\<)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'LESSEQ "<=")

(loop (+ i 2)))

(begin

(add-token 'LESS "<")

(loop (add1 i))))]

[(char=? c #\>)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'GREATEQ ">=")

(loop (+ i 2)))

(begin

(add-token 'GREATER ">")

(loop (add1 i))))]

[(char=? c #\!)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'NOTEQ "!=")

(loop (+ i 2)))

(begin

(add-token 'ERROR "!")

(loop (add1 i))))]

;; Special characters

[(char=? c #\;) (add-token 'SEMICOLON ";") (loop (add1 i))]

[(char=? c #\() (add-token 'LPAREN "(") (loop (add1 i))]

[(char=? c #\)) (add-token 'RPAREN ")") (loop (add1 i))]

[(char=? c #\$)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\$))

(begin

(add-token 'ENDMARKER "$$")

(loop (+ i 2)))

(begin

(add-token 'ERROR "$")

(loop (add1 i))))]

;; Invalid character

[else

(add-token 'ERROR (string c))

(loop (add1 i))]))))

(values tokens line-positions))

;; Parser

(define (parse filename)

(define input (file->string filename))

(define-values (tokens line-positions) (tokenize input))

(define current-token-index 0)

(define parse-error #f)

;; Look at current token without consuming it

(define (peek-token)

(if (< current-token-index (length tokens))

(list-ref tokens current-token-index)

(list 'EOF "" 0)))

;; Consume and return the current token

(define (consume-token)

(let ((token (peek-token)))

(set! current-token-index (add1 current-token-index))

token))

;; Match a specific token type

(define (match-token expected-type)

(let ((token (peek-token)))

(if (eq? (first token) expected-type)

(consume-token)

(begin

(set! parse-error (format "Syntax error at line ~a: expected ~a but got ~a"

(third token)

expected-type

(first token)))

#f))))

;; Get the line content for error reporting

(define (get-line-content line-number)

(let loop ((line-pos-pairs line-positions)

(start 0)

(end (string-length input)))

(cond

[(null? line-pos-pairs) (substring input start end)]

[(= (caar line-pos-pairs) line-number)

(substring input start (cdar line-pos-pairs))]

[else (loop (cdr line-pos-pairs) (cdar line-pos-pairs) end)])))

;; Grammar implementation - recursive descent functions

;; Forward declarations to avoid unbound identifier errors

(define parse-expr #f)

(define parse-stmt #f)

(define parse-stmt-list #f)

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

(define (parse-program)

(let ((result (parse-stmt-list)))

(if (and result (match-token 'ENDMARKER))

(list 'PROGRAM result)

#f)))

;; num -> numsign digit digit\*

(define (parse-num)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "+" (second num))))

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "-" (second num))))

#f))]

[(eq? (first token) 'NUM)

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (second num)))

#f))]

[else #f])))

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

(define (parse-compare)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'LESS)

(consume-token)

'<]

[(eq? (first token) 'LESSEQ)

(consume-token)

'<=]

[(eq? (first token) 'GREATER)

(consume-token)

'>]

[(eq? (first token) 'GREATEQ)

(consume-token)

'>=]

[(eq? (first token) 'EQUALS)

(consume-token)

'==]

[(eq? (first token) 'NOTEQ)

(consume-token)

'!=]

[else #f])))

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

(define (parse-etail)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL '+ expr-result)

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL '- expr-result)

#f))]

[(or (eq? (first token) 'LESS)

(eq? (first token) 'LESSEQ)

(eq? (first token) 'GREATER)

(eq? (first token) 'GREATEQ)

(eq? (first token) 'EQUALS)

(eq? (first token) 'NOTEQ))

(let ((compare (parse-compare)))

(if compare

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL compare expr-result)

#f))

#f))]

[else '()]))) ; epsilon case

;; Set the recursive functions after they've been declared

(set! parse-expr

(lambda ()

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if id

(let ((etail (parse-etail)))

(if etail

(list 'EXPR (list 'ID (second id)) etail)

(list 'EXPR (list 'ID (second id)))))

#f))]

[(or (eq? (first token) 'NUM)

(eq? (first token) 'PLUS)

(eq? (first token) 'MINUS))

(let ((num (parse-num)))

(if num

(let ((etail (parse-etail)))

(if etail

(list 'EXPR num etail)

(list 'EXPR num)))

#f))]

[else #f]))))

(set! parse-stmt

(lambda ()

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if (and id (match-token 'ASSIGN))

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(list 'ASSIGN\_STMT (second id) expr)

#f))

#f))]

[(eq? (first token) 'IF)

(consume-token) ; consume IF

(if (match-token 'LPAREN)

(let ((expr (parse-expr)))

(if (and expr (match-token 'RPAREN))

(let ((stmt-list (parse-stmt-list)))

(if (and stmt-list

(match-token 'ENDIF)

(match-token 'SEMICOLON))

(list 'IF\_STMT expr stmt-list)

#f))

#f))

#f)]

[(eq? (first token) 'READ)

(consume-token) ; consume READ

(let ((id (match-token 'ID)))

(if (and id (match-token 'SEMICOLON))

(list 'READ\_STMT (second id))

#f))]

[(eq? (first token) 'WRITE)

(consume-token) ; consume WRITE

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(list 'WRITE\_STMT expr)

#f))]

[else #f]))))

(set! parse-stmt-list

(lambda ()

(let ((token (peek-token)))

(if (or (eq? (first token) 'ID)

(eq? (first token) 'IF)

(eq? (first token) 'READ)

(eq? (first token) 'WRITE))

(let ((stmt (parse-stmt)))

(if stmt

(let ((stmt-list (parse-stmt-list)))

(if stmt-list

(list 'STMT\_LIST stmt stmt-list)

(list 'STMT\_LIST stmt)))

(list 'STMT\_LIST))) ; error in stmt, return empty list

(list 'STMT\_LIST))))) ; epsilon case

;; Start parsing

(let ((parse-tree (parse-program)))

(if parse-error

(let\* ((error-token (peek-token))

(line-number (third error-token))

(line-content (get-line-content line-number)))

(string-append parse-error "\nLine " (number->string line-number) ": " line-content))

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

;; Main function - exports the parse function for use

(provide parse)

1. **substring: ending index is out of range ending index: 93 valid range: [0, 91] string: "A = 6; \r\nB = 5; \r\nX = -2; \r\nif (A + B > 10)\r\n X = A + 2;\r\nendif \r\nwrite A + B - X; \r\n $$ "**

**>**

#lang racket

;; Lexical analyzer

(define (tokenize input-string)

(define tokens '())

(define current-line 1)

(define line-positions '())

(define (add-token type value)

(set! tokens (append tokens (list (list type value current-line)))))

(define (add-line-position pos)

(set! line-positions (append line-positions (list (cons current-line pos)))))

;; Add initial line position for first line

(add-line-position 0)

(let loop ((i 0))

(when (< i (string-length input-string))

(let ((c (string-ref input-string i)))

(cond

;; New line

[(or (char=? c #\newline) (char=? c #\return))

(when (or (not (< (add1 i) (string-length input-string)))

(not (and (char=? c #\return)

(char=? (string-ref input-string (add1 i)) #\newline))))

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

(add-line-position (add1 i)))

(loop (add1 i))]

;; Whitespace

[(char-whitespace? c)

(loop (add1 i))]

;; Keywords and identifiers

[(char-alphabetic? c)

(let id-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-alphabetic? (string-ref input-string j)))

(id-loop (add1 j))

(let ((word (substring input-string i j)))

(cond

[(string=? word "if") (add-token 'IF word)]

[(string=? word "endif") (add-token 'ENDIF word)]

[(string=? word "read") (add-token 'READ word)]

[(string=? word "write") (add-token 'WRITE word)]

[else (add-token 'ID word)])

(loop j))))]

;; Numbers: digits

[(char-numeric? c)

(let num-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-numeric? (string-ref input-string j)))

(num-loop (add1 j))

(begin

(add-token 'NUM (substring input-string i j))

(loop j))))]

;; Signs and operators

[(char=? c #\+) (add-token 'PLUS "+") (loop (add1 i))]

[(char=? c #\-) (add-token 'MINUS "-") (loop (add1 i))]

[(char=? c #\=)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'EQUALS "==")

(loop (+ i 2)))

(begin

(add-token 'ASSIGN "=")

(loop (add1 i))))]

[(char=? c #\<)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'LESSEQ "<=")

(loop (+ i 2)))

(begin

(add-token 'LESS "<")

(loop (add1 i))))]

[(char=? c #\>)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'GREATEQ ">=")

(loop (+ i 2)))

(begin

(add-token 'GREATER ">")

(loop (add1 i))))]

[(char=? c #\!)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'NOTEQ "!=")

(loop (+ i 2)))

(begin

(add-token 'ERROR "!")

(loop (add1 i))))]

;; Special characters

[(char=? c #\;) (add-token 'SEMICOLON ";") (loop (add1 i))]

[(char=? c #\() (add-token 'LPAREN "(") (loop (add1 i))]

[(char=? c #\)) (add-token 'RPAREN ")") (loop (add1 i))]

[(char=? c #\$)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\$))

(begin

(add-token 'ENDMARKER "$$")

(loop (+ i 2)))

(begin

(add-token 'ERROR "$")

(loop (add1 i))))]

;; Invalid character

[else

(add-token 'ERROR (string c))

(loop (add1 i))]))))

(values tokens line-positions))

;; Parse a line of code from a given file

(define (get-line-at-position input line-positions line-number)

(define start-pos

(cond

[(= line-number 1) 0]

[(> line-number 1)

(let loop ([i 0])

(cond

[(>= i (length line-positions)) 0]

[(= (caar line-positions) (sub1 line-number)) (cdar line-positions)]

[(= (caar line-positions) line-number) (cdar line-positions)]

[else (loop (add1 i))]))]

[else 0]))

(define end-pos

(let loop ([i 0])

(cond

[(>= i (length line-positions)) (string-length input)]

[(= (caar line-positions) line-number) (cdar line-positions)]

[else (loop (add1 i))])))

(if (< start-pos (string-length input))

(substring input start-pos (min end-pos (string-length input)))

""))

;; Parser

(define (parse filename)

(define input (file->string filename))

(define-values (tokens line-positions) (tokenize input))

(define current-token-index 0)

(define parse-error #f)

;; Look at current token without consuming it

(define (peek-token)

(if (< current-token-index (length tokens))

(list-ref tokens current-token-index)

(list 'EOF "" 0)))

;; Consume and return the current token

(define (consume-token)

(let ((token (peek-token)))

(set! current-token-index (add1 current-token-index))

token))

;; Match a specific token type

(define (match-token expected-type)

(let ((token (peek-token)))

(if (eq? (first token) expected-type)

(consume-token)

(begin

(set! parse-error (format "Syntax error at line ~a: expected ~a but got ~a"

(third token)

expected-type

(first token)))

#f))))

;; Forward declarations for mutually recursive functions

(define parse-expr #f)

(define parse-stmt #f)

(define parse-stmt-list #f)

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

(define (parse-program)

(let ((result (parse-stmt-list)))

(if (and result (match-token 'ENDMARKER))

(list 'PROGRAM result)

#f)))

;; num -> numsign digit digit\*

(define (parse-num)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "+" (second num))))

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "-" (second num))))

#f))]

[(eq? (first token) 'NUM)

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (second num)))

#f))]

[else #f])))

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

(define (parse-compare)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'LESS)

(consume-token)

'<]

[(eq? (first token) 'LESSEQ)

(consume-token)

'<=]

[(eq? (first token) 'GREATER)

(consume-token)

'>]

[(eq? (first token) 'GREATEQ)

(consume-token)

'>=]

[(eq? (first token) 'EQUALS)

(consume-token)

'==]

[(eq? (first token) 'NOTEQ)

(consume-token)

'!=]

[else #f])))

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

(define (parse-etail)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL '+ expr-result)

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL '- expr-result)

#f))]

[(or (eq? (first token) 'LESS)

(eq? (first token) 'LESSEQ)

(eq? (first token) 'GREATER)

(eq? (first token) 'GREATEQ)

(eq? (first token) 'EQUALS)

(eq? (first token) 'NOTEQ))

(let ((compare (parse-compare)))

(if compare

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL compare expr-result)

#f))

#f))]

[else '()]))) ; epsilon case

;; Set the recursive functions after they've been declared

(set! parse-expr

(lambda ()

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if id

(let ((etail (parse-etail)))

(if (and etail (not (null? etail)))

(list 'EXPR (list 'ID (second id)) etail)

(list 'EXPR (list 'ID (second id)))))

#f))]

[(or (eq? (first token) 'NUM)

(eq? (first token) 'PLUS)

(eq? (first token) 'MINUS))

(let ((num (parse-num)))

(if num

(let ((etail (parse-etail)))

(if (and etail (not (null? etail)))

(list 'EXPR num etail)

(list 'EXPR num)))

#f))]

[else #f]))))

(set! parse-stmt

(lambda ()

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if (and id (match-token 'ASSIGN))

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(list 'ASSIGN\_STMT (second id) expr)

#f))

#f))]

[(eq? (first token) 'IF)

(consume-token) ; consume IF

(if (match-token 'LPAREN)

(let ((expr (parse-expr)))

(if (and expr (match-token 'RPAREN))

(let ((stmt-list (parse-stmt-list)))

(if (and stmt-list

(match-token 'ENDIF)

(match-token 'SEMICOLON))

(list 'IF\_STMT expr stmt-list)

#f))

#f))

#f)]

[(eq? (first token) 'READ)

(consume-token) ; consume READ

(let ((id (match-token 'ID)))

(if (and id (match-token 'SEMICOLON))

(list 'READ\_STMT (second id))

#f))]

[(eq? (first token) 'WRITE)

(consume-token) ; consume WRITE

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(list 'WRITE\_STMT expr)

#f))]

[else #f]))))

(set! parse-stmt-list

(lambda ()

(let ((token (peek-token)))

(if (or (eq? (first token) 'ID)

(eq? (first token) 'IF)

(eq? (first token) 'READ)

(eq? (first token) 'WRITE))

(let ((stmt (parse-stmt)))

(if stmt

(let ((stmt-list (parse-stmt-list)))

(if stmt-list

(list 'STMT\_LIST stmt stmt-list)

(list 'STMT\_LIST stmt)))

(list 'STMT\_LIST))) ; error in stmt, return empty list

(list 'STMT\_LIST))))) ; epsilon case

;; Start parsing

(let ((parse-tree (parse-program)))

(if parse-error

(let\* ((error-token (peek-token))

(line-number (third error-token))

(line-content (get-line-at-position input line-positions line-number)))

(string-append parse-error "\nLine " (number->string line-number) ": " line-content))

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

;; Main function - exports the parse function for use

(provide parse)

1. **This code works but I need to do a few more things, the outputs should be printed to an output file, it should include a parse tree of the input file ( the parse tree will be a large tree structure contained in a list.) along with progress messages will probably be helpful in development. then at the end print the final message**

#lang racket

;; Lexical analyzer

(define (tokenize input-string)

(printf "Starting lexical analysis...\n")

(define tokens '())

(define current-line 1)

(define line-positions '())

(define (add-token type value)

(set! tokens (append tokens (list (list type value current-line)))))

(define (add-line-position pos)

(set! line-positions (append line-positions (list (cons current-line pos)))))

;; Add initial line position for first line

(add-line-position 0)

(let loop ((i 0))

(when (< i (string-length input-string))

(let ((c (string-ref input-string i)))

(cond

;; New line

[(or (char=? c #\newline) (char=? c #\return))

(when (or (not (< (add1 i) (string-length input-string)))

(not (and (char=? c #\return)

(char=? (string-ref input-string (add1 i)) #\newline))))

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

(add-line-position (add1 i)))

(loop (add1 i))]

;; Whitespace

[(char-whitespace? c)

(loop (add1 i))]

;; Keywords and identifiers

[(char-alphabetic? c)

(let id-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-alphabetic? (string-ref input-string j)))

(id-loop (add1 j))

(let ((word (substring input-string i j)))

(cond

[(string=? word "if") (add-token 'IF word)]

[(string=? word "endif") (add-token 'ENDIF word)]

[(string=? word "read") (add-token 'READ word)]

[(string=? word "write") (add-token 'WRITE word)]

[else (add-token 'ID word)])

(loop j))))]

;; Numbers: digits

[(char-numeric? c)

(let num-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-numeric? (string-ref input-string j)))

(num-loop (add1 j))

(begin

(add-token 'NUM (substring input-string i j))

(loop j))))]

;; Signs and operators

[(char=? c #\+) (add-token 'PLUS "+") (loop (add1 i))]

[(char=? c #\-) (add-token 'MINUS "-") (loop (add1 i))]

[(char=? c #\=)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'EQUALS "==")

(loop (+ i 2)))

(begin

(add-token 'ASSIGN "=")

(loop (add1 i))))]

[(char=? c #\<)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'LESSEQ "<=")

(loop (+ i 2)))

(begin

(add-token 'LESS "<")

(loop (add1 i))))]

[(char=? c #\>)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'GREATEQ ">=")

(loop (+ i 2)))

(begin

(add-token 'GREATER ">")

(loop (add1 i))))]

[(char=? c #\!)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'NOTEQ "!=")

(loop (+ i 2)))

(begin

(add-token 'ERROR "!")

(loop (add1 i))))]

;; Special characters

[(char=? c #\;) (add-token 'SEMICOLON ";") (loop (add1 i))]

[(char=? c #\() (add-token 'LPAREN "(") (loop (add1 i))]

[(char=? c #\)) (add-token 'RPAREN ")") (loop (add1 i))]

[(char=? c #\$)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\$))

(begin

(add-token 'ENDMARKER "$$")

(loop (+ i 2)))

(begin

(add-token 'ERROR "$")

(loop (add1 i))))]

;; Invalid character

[else

(add-token 'ERROR (string c))

(loop (add1 i))]))))

(printf "Lexical analysis complete. Found ~a tokens.\n" (length tokens))

(values tokens line-positions))

;; Parse a line of code from a given file

(define (get-line-at-position input line-positions line-number)

(define start-pos

(cond

[(= line-number 1) 0]

[(> line-number 1)

(let loop ([i 0])

(cond

[(>= i (length line-positions)) 0]

[(= (caar line-positions) (sub1 line-number)) (cdar line-positions)]

[(= (caar line-positions) line-number) (cdar line-positions)]

[else (loop (add1 i))]))]

[else 0]))

(define end-pos

(let loop ([i 0])

(cond

[(>= i (length line-positions)) (string-length input)]

[(= (caar line-positions) line-number) (cdar line-positions)]

[else (loop (add1 i))])))

(if (< start-pos (string-length input))

(substring input start-pos (min end-pos (string-length input)))

""))

;; Format parse tree with proper indentation for better readability

(define (format-parse-tree tree [indent 0])

(define indent-str (make-string (\* indent 2) #\space))

(cond

[(null? tree) ""]

[(pair? tree)

(let ([head (car tree)]

[tail (cdr tree)])

(cond

[(pair? head)

(string-append indent-str "(\n"

(format-parse-tree head (add1 indent)) "\n"

(format-parse-tree tail (add1 indent)) "\n"

indent-str ")")]

[(symbol? head)

(string-append indent-str "(" (symbol->string head) "\n"

(format-parse-tree tail (add1 indent)) "\n"

indent-str ")")]

[else

(string-append indent-str (format "~a" head) "\n"

(format-parse-tree tail indent))]))]

[(symbol? tree)

(string-append indent-str (symbol->string tree))]

[else

(string-append indent-str (format "~a" tree))]))

;; Parser

(define (parse filename)

(printf "Starting to parse file: ~a\n" filename)

(define output-filename (string-append filename ".out"))

(define output-port (open-output-file output-filename #:exists 'replace))

(define (write-output str)

(display str output-port)

(newline output-port))

(define (print-progress msg)

(printf "~a\n" msg)

(write-output msg))

(print-progress (format "Processing file: ~a" filename))

(when (not (file-exists? filename))

(print-progress (format "Error: File not found: ~a" filename))

(close-output-port output-port)

(error (format "File not found: ~a" filename)))

(define input (file->string filename))

(print-progress "Starting lexical analysis...")

(define-values (tokens line-positions) (tokenize input))

(print-progress (format "Lexical analysis complete. Found ~a tokens." (length tokens)))

(define current-token-index 0)

(define parse-error #f)

;; Look at current token without consuming it

(define (peek-token)

(if (< current-token-index (length tokens))

(list-ref tokens current-token-index)

(list 'EOF "" 0)))

;; Consume and return the current token

(define (consume-token)

(let ((token (peek-token)))

(set! current-token-index (add1 current-token-index))

token))

;; Match a specific token type

(define (match-token expected-type)

(let ((token (peek-token)))

(if (eq? (first token) expected-type)

(consume-token)

(begin

(set! parse-error (format "Syntax error at line ~a: expected ~a but got ~a"

(third token)

expected-type

(first token)))

#f))))

;; Forward declarations for mutually recursive functions

(define parse-expr #f)

(define parse-stmt #f)

(define parse-stmt-list #f)

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

(define (parse-program)

(print-progress "Parsing program...")

(let ((result (parse-stmt-list)))

(if (and result (match-token 'ENDMARKER))

(begin

(print-progress "Program parsed successfully.")

(list 'PROGRAM result))

(begin

(print-progress "Error parsing program.")

#f))))

;; num -> numsign digit digit\*

(define (parse-num)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "+" (second num))))

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "-" (second num))))

#f))]

[(eq? (first token) 'NUM)

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (second num)))

#f))]

[else #f])))

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

(define (parse-compare)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'LESS)

(consume-token)

'<]

[(eq? (first token) 'LESSEQ)

(consume-token)

'<=]

[(eq? (first token) 'GREATER)

(consume-token)

'>]

[(eq? (first token) 'GREATEQ)

(consume-token)

'>=]

[(eq? (first token) 'EQUALS)

(consume-token)

'==]

[(eq? (first token) 'NOTEQ)

(consume-token)

'!=]

[else #f])))

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

(define (parse-etail)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL '+ expr-result)

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL '- expr-result)

#f))]

[(or (eq? (first token) 'LESS)

(eq? (first token) 'LESSEQ)

(eq? (first token) 'GREATER)

(eq? (first token) 'GREATEQ)

(eq? (first token) 'EQUALS)

(eq? (first token) 'NOTEQ))

(let ((compare (parse-compare)))

(if compare

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL compare expr-result)

#f))

#f))]

[else '()]))) ; epsilon case

;; Set the recursive functions after they've been declared

(set! parse-expr

(lambda ()

(print-progress "Parsing expression...")

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if id

(let ((etail (parse-etail)))

(if (and etail (not (null? etail)))

(list 'EXPR (list 'ID (second id)) etail)

(list 'EXPR (list 'ID (second id)))))

#f))]

[(or (eq? (first token) 'NUM)

(eq? (first token) 'PLUS)

(eq? (first token) 'MINUS))

(let ((num (parse-num)))

(if num

(let ((etail (parse-etail)))

(if (and etail (not (null? etail)))

(list 'EXPR num etail)

(list 'EXPR num)))

#f))]

[else #f]))))

(set! parse-stmt

(lambda ()

(print-progress "Parsing statement...")

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if (and id (match-token 'ASSIGN))

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(begin

(print-progress "Parsed assignment statement.")

(list 'ASSIGN\_STMT (second id) expr))

#f))

#f))]

[(eq? (first token) 'IF)

(consume-token) ; consume IF

(print-progress "Parsing IF statement...")

(if (match-token 'LPAREN)

(let ((expr (parse-expr)))

(if (and expr (match-token 'RPAREN))

(let ((stmt-list (parse-stmt-list)))

(if (and stmt-list

(match-token 'ENDIF)

(match-token 'SEMICOLON))

(begin

(print-progress "Parsed IF statement.")

(list 'IF\_STMT expr stmt-list))

#f))

#f))

#f)]

[(eq? (first token) 'READ)

(consume-token) ; consume READ

(print-progress "Parsing READ statement...")

(let ((id (match-token 'ID)))

(if (and id (match-token 'SEMICOLON))

(begin

(print-progress "Parsed READ statement.")

(list 'READ\_STMT (second id)))

#f))]

[(eq? (first token) 'WRITE)

(consume-token) ; consume WRITE

(print-progress "Parsing WRITE statement...")

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(begin

(print-progress "Parsed WRITE statement.")

(list 'WRITE\_STMT expr))

#f))]

[else #f]))))

(set! parse-stmt-list

(lambda ()

(print-progress "Parsing statement list...")

(let ((token (peek-token)))

(if (or (eq? (first token) 'ID)

(eq? (first token) 'IF)

(eq? (first token) 'READ)

(eq? (first token) 'WRITE))

(let ((stmt (parse-stmt)))

(if stmt

(let ((stmt-list (parse-stmt-list)))

(if stmt-list

(list 'STMT\_LIST stmt stmt-list)

(list 'STMT\_LIST stmt)))

(list 'STMT\_LIST))) ; error in stmt, return empty list

(begin

(print-progress "End of statement list.")

(list 'STMT\_LIST)))))) ; epsilon case

;; Start parsing

(let ((parse-tree (parse-program)))

(if parse-error

(let\* ((error-token (peek-token))

(line-number (third error-token))

(line-content (get-line-at-position input line-positions line-number))

(error-message (string-append parse-error "\nLine " (number->string line-number) ": " line-content)))

(print-progress (format "Parse Error: ~a" error-message))

(print-progress "Parsing failed.")

(close-output-port output-port)

error-message)

(let ((result (format "Accept\n~a" parse-tree))

(formatted-tree (format-parse-tree parse-tree)))

(print-progress "Parsing successful.")

(print-progress "Parse Tree:")

(write-output formatted-tree)

(print-progress "Final Result: Accept")

(close-output-port output-port)

(printf "Results written to: ~a\n" output-filename)

result))))

;; Main function

(define (process-files filenames)

(printf "Starting to process ~a files...\n" (length filenames))

(for ([filename filenames]

[i (in-naturals 1)])

(printf "\nProcessing file ~a/~a: ~a\n" i (length filenames) filename)

(parse filename))

(printf "\nAll files processed.\n"))

;; Export the parse function for use

(provide parse process-files)

;; Process files

(define files-to-process '("file1.txt" "file2.txt" "file3.txt" "file4.txt"))

(process-files files-to-process)

1. **Instead of using a. Out extension just rename the output files as output 1-4**

#lang racket

;; Lexical analyzer

(define (tokenize input-string)

(printf "Starting lexical analysis...\n")

(define tokens '())

(define current-line 1)

(define line-positions '())

(define (add-token type value)

(set! tokens (append tokens (list (list type value current-line)))))

(define (add-line-position pos)

(set! line-positions (append line-positions (list (cons current-line pos)))))

;; Add initial line position for first line

(add-line-position 0)

(let loop ((i 0))

(when (< i (string-length input-string))

(let ((c (string-ref input-string i)))

(cond

;; New line

[(or (char=? c #\newline) (char=? c #\return))

(when (or (not (< (add1 i) (string-length input-string)))

(not (and (char=? c #\return)

(char=? (string-ref input-string (add1 i)) #\newline))))

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

(add-line-position (add1 i)))

(loop (add1 i))]

;; Whitespace

[(char-whitespace? c)

(loop (add1 i))]

;; Keywords and identifiers

[(char-alphabetic? c)

(let id-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-alphabetic? (string-ref input-string j)))

(id-loop (add1 j))

(let ((word (substring input-string i j)))

(cond

[(string=? word "if") (add-token 'IF word)]

[(string=? word "endif") (add-token 'ENDIF word)]

[(string=? word "read") (add-token 'READ word)]

[(string=? word "write") (add-token 'WRITE word)]

[else (add-token 'ID word)])

(loop j))))]

;; Numbers: digits

[(char-numeric? c)

(let num-loop ((j (add1 i)))

(if (and (< j (string-length input-string))

(char-numeric? (string-ref input-string j)))

(num-loop (add1 j))

(begin

(add-token 'NUM (substring input-string i j))

(loop j))))]

;; Signs and operators

[(char=? c #\+) (add-token 'PLUS "+") (loop (add1 i))]

[(char=? c #\-) (add-token 'MINUS "-") (loop (add1 i))]

[(char=? c #\=)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'EQUALS "==")

(loop (+ i 2)))

(begin

(add-token 'ASSIGN "=")

(loop (add1 i))))]

[(char=? c #\<)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'LESSEQ "<=")

(loop (+ i 2)))

(begin

(add-token 'LESS "<")

(loop (add1 i))))]

[(char=? c #\>)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'GREATEQ ">=")

(loop (+ i 2)))

(begin

(add-token 'GREATER ">")

(loop (add1 i))))]

[(char=? c #\!)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\=))

(begin

(add-token 'NOTEQ "!=")

(loop (+ i 2)))

(begin

(add-token 'ERROR "!")

(loop (add1 i))))]

;; Special characters

[(char=? c #\;) (add-token 'SEMICOLON ";") (loop (add1 i))]

[(char=? c #\() (add-token 'LPAREN "(") (loop (add1 i))]

[(char=? c #\)) (add-token 'RPAREN ")") (loop (add1 i))]

[(char=? c #\$)

(if (and (< (add1 i) (string-length input-string))

(char=? (string-ref input-string (add1 i)) #\$))

(begin

(add-token 'ENDMARKER "$$")

(loop (+ i 2)))

(begin

(add-token 'ERROR "$")

(loop (add1 i))))]

;; Invalid character

[else

(add-token 'ERROR (string c))

(loop (add1 i))]))))

(printf "Lexical analysis complete. Found ~a tokens.\n" (length tokens))

(values tokens line-positions))

;; Parse a line of code from a given file

(define (get-line-at-position input line-positions line-number)

(define start-pos

(cond

[(= line-number 1) 0]

[(> line-number 1)

(let loop ([i 0])

(cond

[(>= i (length line-positions)) 0]

[(= (caar line-positions) (sub1 line-number)) (cdar line-positions)]

[(= (caar line-positions) line-number) (cdar line-positions)]

[else (loop (add1 i))]))]

[else 0]))

(define end-pos

(let loop ([i 0])

(cond

[(>= i (length line-positions)) (string-length input)]

[(= (caar line-positions) line-number) (cdar line-positions)]

[else (loop (add1 i))])))

(if (< start-pos (string-length input))

(substring input start-pos (min end-pos (string-length input)))

""))

;; Format parse tree with proper indentation for better readability

(define (format-parse-tree tree [indent 0])

(define indent-str (make-string (\* indent 2) #\space))

(cond

[(null? tree) ""]

[(pair? tree)

(let ([head (car tree)]

[tail (cdr tree)])

(cond

[(pair? head)

(string-append indent-str "(\n"

(format-parse-tree head (add1 indent)) "\n"

(format-parse-tree tail (add1 indent)) "\n"

indent-str ")")]

[(symbol? head)

(string-append indent-str "(" (symbol->string head) "\n"

(format-parse-tree tail (add1 indent)) "\n"

indent-str ")")]

[else

(string-append indent-str (format "~a" head) "\n"

(format-parse-tree tail indent))]))]

[(symbol? tree)

(string-append indent-str (symbol->string tree))]

[else

(string-append indent-str (format "~a" tree))]))

;; Parser

(define (parse filename file-index)

(printf "Starting to parse file: ~a\n" filename)

(define output-filename (format "output~a.txt" file-index))

(define output-port (open-output-file output-filename #:exists 'replace))

(define (write-output str)

(display str output-port)

(newline output-port))

(define (print-progress msg)

(printf "~a\n" msg)

(write-output msg))

(print-progress (format "Processing file: ~a" filename))

(when (not (file-exists? filename))

(print-progress (format "Error: File not found: ~a" filename))

(close-output-port output-port)

(error (format "File not found: ~a" filename)))

(define input (file->string filename))

(print-progress "Starting lexical analysis...")

(define-values (tokens line-positions) (tokenize input))

(print-progress (format "Lexical analysis complete. Found ~a tokens." (length tokens)))

(define current-token-index 0)

(define parse-error #f)

;; Look at current token without consuming it

(define (peek-token)

(if (< current-token-index (length tokens))

(list-ref tokens current-token-index)

(list 'EOF "" 0)))

;; Consume and return the current token

(define (consume-token)

(let ((token (peek-token)))

(set! current-token-index (add1 current-token-index))

token))

;; Match a specific token type

(define (match-token expected-type)

(let ((token (peek-token)))

(if (eq? (first token) expected-type)

(consume-token)

(begin

(set! parse-error (format "Syntax error at line ~a: expected ~a but got ~a"

(third token)

expected-type

(first token)))

#f))))

;; Forward declarations for mutually recursive functions

(define parse-expr #f)

(define parse-stmt #f)

(define parse-stmt-list #f)

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

(define (parse-program)

(print-progress "Parsing program...")

(let ((result (parse-stmt-list)))

(if (and result (match-token 'ENDMARKER))

(begin

(print-progress "Program parsed successfully.")

(list 'PROGRAM result))

(begin

(print-progress "Error parsing program.")

#f))))

;; num -> numsign digit digit\*

(define (parse-num)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "+" (second num))))

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (string-append "-" (second num))))

#f))]

[(eq? (first token) 'NUM)

(let ((num (match-token 'NUM)))

(if num

(list 'NUM (string->number (second num)))

#f))]

[else #f])))

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

(define (parse-compare)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'LESS)

(consume-token)

'<]

[(eq? (first token) 'LESSEQ)

(consume-token)

'<=]

[(eq? (first token) 'GREATER)

(consume-token)

'>]

[(eq? (first token) 'GREATEQ)

(consume-token)

'>=]

[(eq? (first token) 'EQUALS)

(consume-token)

'==]

[(eq? (first token) 'NOTEQ)

(consume-token)

'!=]

[else #f])))

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

(define (parse-etail)

(let ((token (peek-token)))

(cond

[(eq? (first token) 'PLUS)

(consume-token) ; consume +

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL '+ expr-result)

#f))]

[(eq? (first token) 'MINUS)

(consume-token) ; consume -

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL '- expr-result)

#f))]

[(or (eq? (first token) 'LESS)

(eq? (first token) 'LESSEQ)

(eq? (first token) 'GREATER)

(eq? (first token) 'GREATEQ)

(eq? (first token) 'EQUALS)

(eq? (first token) 'NOTEQ))

(let ((compare (parse-compare)))

(if compare

(let ((expr-result (parse-expr)))

(if expr-result

(list 'ETAIL compare expr-result)

#f))

#f))]

[else '()]))) ; epsilon case

;; Set the recursive functions after they've been declared

(set! parse-expr

(lambda ()

(print-progress "Parsing expression...")

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if id

(let ((etail (parse-etail)))

(if (and etail (not (null? etail)))

(list 'EXPR (list 'ID (second id)) etail)

(list 'EXPR (list 'ID (second id)))))

#f))]

[(or (eq? (first token) 'NUM)

(eq? (first token) 'PLUS)

(eq? (first token) 'MINUS))

(let ((num (parse-num)))

(if num

(let ((etail (parse-etail)))

(if (and etail (not (null? etail)))

(list 'EXPR num etail)

(list 'EXPR num)))

#f))]

[else #f]))))

(set! parse-stmt

(lambda ()

(print-progress "Parsing statement...")

(let ((token (peek-token)))

(cond

[(eq? (first token) 'ID)

(let ((id (match-token 'ID)))

(if (and id (match-token 'ASSIGN))

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(begin

(print-progress "Parsed assignment statement.")

(list 'ASSIGN\_STMT (second id) expr))

#f))

#f))]

[(eq? (first token) 'IF)

(consume-token) ; consume IF

(print-progress "Parsing IF statement...")

(if (match-token 'LPAREN)

(let ((expr (parse-expr)))

(if (and expr (match-token 'RPAREN))

(let ((stmt-list (parse-stmt-list)))

(if (and stmt-list

(match-token 'ENDIF)

(match-token 'SEMICOLON))

(begin

(print-progress "Parsed IF statement.")

(list 'IF\_STMT expr stmt-list))

#f))

#f))

#f)]

[(eq? (first token) 'READ)

(consume-token) ; consume READ

(print-progress "Parsing READ statement...")

(let ((id (match-token 'ID)))

(if (and id (match-token 'SEMICOLON))

(begin

(print-progress "Parsed READ statement.")

(list 'READ\_STMT (second id)))

#f))]

[(eq? (first token) 'WRITE)

(consume-token) ; consume WRITE

(print-progress "Parsing WRITE statement...")

(let ((expr (parse-expr)))

(if (and expr (match-token 'SEMICOLON))

(begin

(print-progress "Parsed WRITE statement.")

(list 'WRITE\_STMT expr))

#f))]

[else #f]))))

(set! parse-stmt-list

(lambda ()

(print-progress "Parsing statement list...")

(let ((token (peek-token)))

(if (or (eq? (first token) 'ID)

(eq? (first token) 'IF)

(eq? (first token) 'READ)

(eq? (first token) 'WRITE))

(let ((stmt (parse-stmt)))

(if stmt

(let ((stmt-list (parse-stmt-list)))

(if stmt-list

(list 'STMT\_LIST stmt stmt-list)

(list 'STMT\_LIST stmt)))

(list 'STMT\_LIST))) ; error in stmt, return empty list

(begin

(print-progress "End of statement list.")

(list 'STMT\_LIST)))))) ; epsilon case

;; Start parsing

(let ((parse-tree (parse-program)))

(if parse-error

(let\* ((error-token (peek-token))

(line-number (third error-token))

(line-content (get-line-at-position input line-positions line-number))

(error-message (string-append parse-error "\nLine " (number->string line-number) ": " line-content)))

(print-progress (format "Parse Error: ~a" error-message))

(print-progress "Parsing failed.")

(close-output-port output-port)

error-message)

(let ((result (format "Accept\n~a" parse-tree))

(formatted-tree (format-parse-tree parse-tree)))

(print-progress "Parsing successful.")

(print-progress "Parse Tree:")

(write-output formatted-tree)

(print-progress "Final Result: Accept")

(close-output-port output-port)

(printf "Results written to: ~a\n" output-filename)

result))))

;; Main function

(define (process-files filenames)

(printf "Starting to process ~a files...\n" (length filenames))

(for ([filename filenames]

[i (in-naturals 1)])

(printf "\nProcessing file ~a/~a: ~a\n" i (length filenames) filename)

(parse filename i))

(printf "\nAll files processed.\n"))

;; Export the parse function for use

(provide parse process-files)

;; Process files

(define files-to-process '("file1.txt" "file2.txt" "file3.txt" "file4.txt"))

(process-files files-to-process)

1. **with the grammar given create an input file 5 that will work and actually produce the desired result -**

program -> {stmt\_list} $$

stmt\_list -> stmt stmt\_list

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 -> < | <= | > | >= | == | !=

x = 5;

y = 10;

if (x < y)

z = x + y;

write z;

endif;

read a;

if (a == 10)

write a + 5;

endif;

$$