Code reference and modification

Rocket Documentation: [Racket Documentation](https://docs.racket-lang.org/)

LLM: ChatGPT-4

Video link: [Wed Oct 30 2024 9:48:44 PM](https://umsystem.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f067d831-d3c9-4e2c-83c6-b21a003232c3)

I lost all the files when I try to clone to github. luckly, I have most of files in backup and upload them on canvas already, but not with this code reference file. That’s why it cause delay. sorry about that.

Prompt: Write a Racket function that parses a list of tokens into a nested expression tree, handling addition, subtraction, multiplication, and division with descending operator precedence.

ChatGPT-4 :   
#lang racket

; Parse multiplication and division (higher precedence)

(define (parse-mult-div tokens)

(let ([first (car tokens)])

(cond

; Multiplication case

[(and (>= (length tokens) 3) (equal? (cadr tokens) 'mult-op))

(let-values ([(right rest) (parse-mult-div (cddr tokens))])

(values

(list 'mult-expr

(parse-mult-div (list first))

right)

rest))]

; Division case

[(and (>= (length tokens) 3) (equal? (cadr tokens) 'div-op))

(let-values ([(right rest) (parse-mult-div (cddr tokens))])

(values

(list 'div-expr

(parse-mult-div (list first))

right)

rest))]

; No more multiplications/divisions, move to addition/subtraction parsing

[else (parse-add-sub tokens)])))

; Parse addition and subtraction (lower precedence)

(define (parse-add-sub tokens)

(let ([first (car tokens)])

(cond

; Addition case

[(and (>= (length tokens) 3) (equal? (cadr tokens) 'add-op))

(let-values ([(right rest) (parse-add-sub (cddr tokens))])

(values

(list 'add-expr

(parse-add-sub (list first))

right)

rest))]

; Subtraction case

[(and (>= (length tokens) 3) (equal? (cadr tokens) 'sub-op))

(let-values ([(right rest) (parse-add-sub (cddr tokens))])

(values

(list 'sub-expr

(parse-add-sub (list first))

right)

rest))]

; Base case: return single operand (ID or integer constant)

[else

(if (equal? (car first) 'ID)

(values (list 'id (cadr first)) (cdr tokens))

(values (list 'constant (cadr first)) (cdr tokens)))])))

; Main parse function

(define (parse-expression tokens)

(parse-mult-div tokens))

; Example usage

(define tokens '((ID x) (add-op +) (ID y) (mult-op \*) (integer 2)))

(parse-expression tokens)

Prompt: Write a Racket function that optimizes a parse tree by recursively simplifying statement nodes nodes, converting value nodes with identifier and constant sub-nodes to a simpler form. Each tree node should be recursively processed, and any node structure not explicitly matched should be left unchanged.

ChatGPT-4:  
#lang racket

; Function to optimize a parse tree by simplifying STMT and value nodes

(define (optimize-parse-tree tree)

(cond

; Base case: if it's not a list, return it as is

[(not (list? tree)) tree]

; Empty list case

[(empty? tree) '()]

; Otherwise, pattern-match specific tree structures

[else

(match tree

; Optimization for assignment statement nodes (STMT)

[(list 'STMT id assign expr)

(list 'STMT

(optimize-parse-tree id)

(optimize-parse-tree assign)

(optimize-expression expr))]

; Simplify value nodes with identifier

[(list 'value (list 'id name))

(list 'ID name)]

; Simplify value nodes with constant integer

[(list 'value (list 'constant (list 'integer num)))

(list 'integer num)]

; Base case for constant integers directly in the tree

[(list 'constant (list 'integer num))

(list 'integer num)]

; Handle identifiers directly in the tree

[(list 'id name)

(list 'ID name)]

; Default case: recursively optimize all elements in the tree

[else (map optimize-parse-tree tree)])]))

; Helper function for simplifying expressions within STMT nodes

(define (optimize-expression expr)

(match expr

; Example pattern: simplify a basic add expression

[(list 'expr (list 'add-expr left (list 'constant (list 'integer 0))))

(optimize-parse-tree left)]

; If no specific optimizations, return as is

[else (optimize-parse-tree expr)]))

Prompt: The function should simplify any value nodes with id or constant values to basic ID or integer forms. For logical expressions within expr, match patterns like and-expr, compare-expr, and expressions like add-expr, mult-expr to further simplify.

ChatGPT-4:  
#lang racket

; Main optimization function for parse trees

(define (optimize-parse-tree tree)

(cond

; Base case: if it's not a list, return it as is

[(not (list? tree)) tree]

; Empty list case

[(empty? tree) '()]

; Otherwise, pattern-match specific tree structures

[else

(match tree

; Simplify assignment statements (STMT nodes)

[(list 'STMT id assign expr)

(list 'STMT

(optimize-parse-tree id)

(optimize-parse-tree assign)

(optimize-expr expr))]

; Simplify value nodes with identifier

[(list 'value (list 'id name))

(list 'ID name)]

; Simplify value nodes with constant integer

[(list 'value (list 'constant (list 'integer num)))

(list 'integer num)]

; Handle identifiers directly in the tree

[(list 'id name)

(list 'ID name)]

; Default case: recursively optimize all elements in the tree

[else (map optimize-parse-tree tree)])]))

; Helper function to optimize expressions within STMT nodes

(define (optimize-expr expr)

(match expr

; Simplify and-expr containing a not-expr and compare-expr for logical optimization

[(list 'and-expr (list 'not-expr (list 'compare-expr left op right)))

(list 'and-expr

(list 'not-expr (optimize-expr (list 'compare-expr left op right))))]

; Simplify add-expr with addition of a constant zero

[(list 'add-expr left (list 'constant (list 'integer 0)))

(optimize-parse-tree left)]

; Simplify mult-expr with multiplication by one

[(list 'mult-expr left (list 'constant (list 'integer 1)))

(optimize-parse-tree left)]

; Recursively optimize other expressions within `expr`

[else (optimize-parse-tree expr)]))

Prompt: Generate 2 Racket test files. Each file should include different expressions like variable assignments (x := y + 1), arithmetic operations (\*, /, +, -), and control structures (IF, WHILE, PRINT).

ChatGPT-4:  
  
; File 1  
  
DEF calculate(x)

result := x \* 2

IF result > 10 THEN

RETURN result + 5

ENDIF

RETURN result

ENDDEF

x := 7

answer := calculate(x)

PRINT "The answer is " ; answer

END  
  
  
; File 2  
  
counter := 1

sum := 0

WHILE counter <= 5 DO

sum := sum + counter

counter := counter + 1

ENDWHILE

IF sum > 10 THEN

PRINT "Sum is large: " ; sum

ENDIF

END