;;; Jacob Steckel

;;; Dr.Wyatt

;;; CSC345

;;; 30 October 2019

;;; integrate.lisp

;;;=================================================================

;;; BUG STATUS -- NO KNOWN BUGS

;;;=================================================================

;;; integrate

(defun integrate (F V &optional lower upper)

"Integrates F with respect to V, optionally integrating on the domain [LOWER, UPPER]."

(def-integral (indef-integral F V) V lower upper))

;;;=================================================================

;;; indef-integral

(defun indef-integral (F V)

"Indefinitely integrates F with respect to V."

(labels (

;; Indefinite integration helper function

(indef-integral-aux (F V)

(cond

;; Numbers

((number-p F) (make-product F V))

;; Negative

((negative-p F) (make-negative (integrate (make-negative F) V)))

((variable-p F) (integrate (make-power F 1) V))

;; Addition

((sum-p F) (make-sum

(integrate (sum-first-operand F) V)

(integrate (sum-second-operand F) V)))

;; Subtraction

((difference-p F) (make-difference

(integrate (difference-first-operand F) V)

(integrate (difference-second-operand F) V)))

;; Power (n != -1)

((and (power-p F) (not (equal (power-second-operand F) -1))) (make-quotient

(make-power V (make-sum (power-second-operand F) 1))

(make-sum (power-second-operand F) 1)))

;; Power (n = -1)

((and (power-p F) (equal (power-second-operand F) -1) (make-log (power-first-operand F))))

(t nil))))

(cond

;; Simplify multiple negative symbols

((mult-negative-p F) (indef-integral-aux (make-simplified-negative F) V))

((variable-p V) (indef-integral-aux F V))

(t nil))))

;;;=================================================================

;;; def-integral

(defun def-integral (F V lower upper)

"Definitely integrates F with respect to V, on the domain [LOWER, UPPER]."

(cond

;; Return indefinite integral if limits are not numbers

((not (and (number-p lower) (number-p upper))) F)

;; make the difference of the substituted halves

(t (eval (make-difference

(my-replace V upper F)

(my-replace V lower F))))))

;;;=================================================================

;;; my-replace

(defun my-replace (e1 e2 L)

"Returns the list L with element E1 replaced with E2."

(labels (

;; Helper function

(my-replace-aux (e1 e2 L)

(cond

((endp L) L)

;;replacement when the element is found

((equal (first L) e1) (cons e2 (my-replace e1 e2 (rest L))))

;; down a level if the element itself is a list

((listp (first L)) (cons (my-replace e1 e2 (first L)) (my-replace e1 e2 (rest L))))

;; Continue down the list

(t (cons (first L) (my-replace e1 e2 (rest L)))))))

(cond

;; If L is a variable, perform replacement on a list containing the variable

((variable-p L) (first (my-replace e1 e2 (list L))))

;; replace

(t (my-replace-aux e1 e2 L)))))

;;;=================================================================

;;; SYMBOLS

;;;=================================================================

(defconstant \*variable-symbols\* '(U V W X Y Z))

(defconstant \*negative-symbol\* '-)

(defconstant \*sum-symbol\* '+)

(defconstant \*difference-symbol\* '-)

(defconstant \*product-symbol\* '\*)

(defconstant \*quotient-symbol\* '/)

(defconstant \*power-symbol\* 'expt)

(defconstant \*log-symbol\* 'log)

;;;=================================================================

;;; SELECTORS – OPERATORS

;;;=================================================================

;;; negative-operator

(defun negative-operator (F) (first F))

;;; sum-operator

(defun sum-operator (F) (first F))

;;; difference-operator

(defun difference-operator (F) (first F))

;;; product-operator

(defun product-operator (F) (first F))

;;; quotient-operator

(defun quotient-operator (F) (first F))

;;; power-operator

(defun power-operator (F) (first F))

;;;=================================================================

;;; SELECTORS – OPERANDS

;;;=================================================================

;;; negative-operand

(defun negative-operand (F) (second F))

;;; sum-first-operand

(defun sum-first-operand (F) (second F))

;;; sum-second-operand

(defun sum-second-operand (F) (third F))

;;; difference-first-operand

(defun difference-first-operand (F) (second F))

;;; difference-second-operand

(defun difference-second-operand (F) (third F))

;;; product-first-operand

(defun product-first-operand (F) (second F))

;;; product-second-operand

(defun product-second-operand (F) (third F))

;;; quotient-first-operand

(defun quotient-first-operand (F) (second F))

;;; quotient-second-operand

(defun quotient-second-operand (F) (third F))

;;; power-first-operand

(defun power-first-operand (F) (second F))

;;; power-second-operand

(defun power-second-operand (F) (third F))

;;;=================================================================

;;; PREDICATES

;;;=================================================================

;;; variable-p

(defun variable-p (F)

"Returns T if F is a variable symbol."

(member F \*variable-symbols\*))

;;;=================================================================

;;; number-p

(defun number-p (F)

"Returns T if F is a number."

(numberp F))

;;;=================================================================

;;; negative-p

(defun negative-p (F)

"Returns T if F is a negative expression."

(cond

;; Check if numerically negative

((and (number-p F) (< F 0)) t)

((number-p F) nil)

((variable-p F) nil)

((difference-p F) nil)

;; Check if expression follows the form (- F)

((and

(equal (negative-operator F) \*negative-symbol\*)

(not (equal (negative-operand F) \*negative-symbol\*))) t)))

;;;=================================================================

;;; mult-negative-p

(defun mult-negative-p (F)

"Returns T if F is a negative expression with multiple negations."

(labels (

;; Helper function

(mult-negative-p-aux (F L)

(cond

;; Return T if L contains just one element at the end

((endp F) (equal (length L) 1))

;; If CAR is the negative symbol, continue CDRing

((equal (first F) \*negative-symbol\*) (mult-negative-p-aux (rest F) L))

;; Otherwise, add CAR to L and continue

(t (mult-negative-p-aux (rest F) (cons (first F) L))))))

(cond

((number-p F) nil)

((variable-p F) nil)

((negative-p F) nil)

((difference-p F) nil)

((not (listp F)) nil)

(t (mult-negative-p-aux F '())))))

;;;=================================================================

;;; sum-p

(defun sum-p (F)

"Returns T if F is a sum expression."

(cond

((number-p F) nil)

((variable-p F) nil)

;; Check if expression follows the form (+ F G)

((and

(equal (sum-operator F) \*sum-symbol\*)

(sum-first-operand F)

(sum-second-operand F)) t)))

;;;=================================================================

;;; difference-p

(defun difference-p (F)

"Returns T if F is a difference expression."

(cond

((number-p F) nil)

((variable-p F) nil)

;; Check if expression follows the form (- F G)

((and

(equal (difference-operator F) \*difference-symbol\*)

(not (equal (difference-first-operand F) \*difference-symbol\*))

(difference-second-operand F)) t)))

;;;=================================================================

;;; power-p

(defun power-p (F)

"Returns T if F is a power expression."

(cond

((number-p F) nil)

((variable-p F) nil)

;; Check if expression follows the form (EXPT V N)

((and

(equal (power-operator F) \*power-symbol\*)

(variable-p (power-first-operand F))

(number-p (power-second-operand F))) t)))

;;;=================================================================

;;; CONSTRUCTORS

;;;=================================================================

;;; make-variable

(defun make-variable (V)

"Constructs a variable expression consisting of V."

V)

;;;=================================================================

;;; make-negative

(defun make-negative (F)

"Constructs an expression which is the negation of F."

(labels (

(make-negative-aux (F)

(cond

;; If F is a number, negate it numerically

((number-p F) (\* -1 F))

;; If F is already a negative, get the negation

((negative-p F) (negative-operand F))

;; Returns the list containing the negative expression

(t (list \*negative-symbol\* F)))))

(cond

;; Simplify expression if necessary

((mult-negative-p F) (make-negative-aux (make-simplified-negative F)))

(t (make-negative-aux F)))))

;;;=================================================================

;;; make-simplified-negative

(defun make-simplified-negative (F)

"Constructs an expression which is the simplified negation of F."

(labels (

(make-simplified-negative-aux (F)

(cond

;; Even number of negative symbols cancel each other out

((equal (mod (length F) 2) 0) (list \*negative-symbol\* (first (last F))))

(t (first (last F))))))

(cond

;; If there are multiple negatives, simplify

((mult-negative-p F) (make-simplified-negative-aux F))

;; If F is a single negative, return F

((negative-p F) F))))

;;;=================================================================

;;; make-sum

(defun make-sum (F G)

"Constructs an expression which is the sum of F and G."

(cond

;; F or G plus 0 is itself

((equal F 0) G)

((equal G 0) F)

;; F or G plus its inverse is 0

((equal F (make-negative G)) 0)

((equal G (make-negative F)) 0)

;; If both are numbers, perform addition

((and (number-p F) (number-p G)) (+ F G))

;; Return the list containing the sum expression

(t (list \*sum-symbol\* F G))))

;;;=================================================================

;;; make-difference

(defun make-difference (F G)

"Constructs an expression which is the difference of F and G."

(cond

;; 0 minus G yields the negation of G

((equal F 0) (make-negative G))

;; F minus 0 yields F

((equal G 0) F)

;; If both are numbers, perform subtraction

((and (number-p F) (number-p G)) (- F G))

((equal F (make-negative G)) (make-sum F G))

;; Return the list containing the difference expression

(t (list \*difference-symbol\* F G))))

;;;=================================================================

;;; make-product

(defun make-product (F G)

"Constructs an expression which is the product of F and G."

(cond

;; F or G time 0 is 0

((equal F 0) 0)

((equal G 0) 0)

;; F or G times 1 is itself

((equal F 1) G)

((equal G 1) F)

;; F or G times -1 is its negation

((equal F -1) (make-negative G))

((equal G -1) (make-negative F))

;; If F and G are both negative, negate both and multiply

((and (negative-p F) (negative-p G)) (make-product (make-negative F) (make-negative G)))

;; If both are numbers, perform multiplication

((and (number-p F) (number-p G)) (\* F G))

;; Return the list containing the product expression

(t (list \*product-symbol\* F G))))

;;;=================================================================

;;; make-quotient

(defun make-quotient (F G)

"Constructs an expression which is the quotient of F and G."

(cond

;; 0 divided by anything is 0

((equal F 0) 0)

;; Division by 0 not allowed

((equal G 0) nil)

;; If both are numbers, perform division

((and (number-p F) (number-p G)) (/ F G))

;; Return the list containing the difference expression

(t (list \*quotient-symbol\* F G))))

;;;=================================================================

;;; make-power

(defun make-power (V N)

"Constructs an expression which is V raised to the Nth power."

(cond

;; If both are numbers, perform exponentiation

((and (number-p V) (numberp N)) (expt V N))

;; Return the list containing the power expression

(t (list \*power-symbol\* V N))))

;;;=================================================================

;;; make-log

(defun make-log (V)

"Constructs an expression which is the mathematical logarithm of V."

(cond

;; If V is a variable, return the list containing the logarithmic expression

((variable-p V) (list \*log-symbol\* V))))