3740 group project By Kegan Rieder and Justin Wolfenden

We dealt with this issue by converting the polynomial to the representation that is easiest for the current task, we would then perform the functions using this representation and if necessary, we would change it back to the original representation before returning the value. You can see this demonstrated through add, when it encounters multiple sparse, it will convert these sparse to dense, it will then take the answer it got from the calculation and convert that to sparse before returning it to the user.

A brief explanation of the implementation of all functions

**Is-sparse?(Polynomial)**

Polynomial is the polynomial you want to see if it is sparse.

It works by first checking if the list representation of the polynomial is an empty list. If so it returns false, meaning its not sparse.

Next it checks if the first item in the list is also a list, if it is it will return true (the list is sparse), if the first item in the list is not a list it returns false (the list is not sparse).

**Make-sparse(polynomial index)**

Polynomial is the list representation of the polynomial that you want to make sparse (list given must be dense). Index should be set to 0 and is used for recursion

It works by first checking if the list is empty, if it is it will just return ()

If the list is not empty it will then check if the next item in the list is 0, if it is 0 it will not add it to the new sparse list, it will then run make\_sparse((cdr polynomial) (add1 index)) in order to check the next item in the list with the corresponding index

if the next item in the list is not 0 it will add a small list consisting of the current number and the index or coeficent of that number to the new list and use recursion make\_sparse((cdr polynomial) (add1 index)) in order to check the next item in the list with the corresponding index

after the list is empty it will return the newly created list.

**To-sparse(Polynomial, index)**

Polynomial is the polynomial you want to change to sparse index is set to 0 for recursion

It will first use is\_sparse?(polynomial) to check if the polynomial is already sparse If it is sparse it will return the polynomial given to it, if it is not it will run make\_sparse(polynomial index) to make it sparse

**Is-dense?(x)**

x is the polynomial you want to see if it is dense.

checks if the first item in the list is also a list, if it is it will return false (the list is not dense), if the first item in the list is not a list it returns true (the list is dense).

**Make-dense(x y)**

X is the polynomial you want to make dense (list given must be sparse), y is set to 0 and used for recursion

First it will check if the list is empty, if it is it will return null

If the list is not empty it will check if y is equal to the second item present int the first list of the sparse list, if they are equal then it will add the first item of the first list to the new list , it will then run make\_dense((cdr x) (+ 1 y)) in order to check the next item in the list with the corresponding y.

If y is not equal to the second item of the first list in sparse then it will add 0 to the new list, it will then run make\_dense(( x) (+ 1 y)) in order to keep y updated to the correct index.

**To-dense(x, y)**

x is the polynomial you want to change to sparse y is set to 0 for recursion

It will first use is\_dense?(x) to check if the polynomial is already sparse If it is dense it will return the polynomial given to it, if it is not it will run make\_dense(x y) to make it dense

**Is-zero?(polynomial)**

Polynomial is the polynomial you want to check if zero, works with both sparse and dense.

First checks if the list is empty, returns true if it is, If the list is not empty, it checks if the list is sparse, if it is sparse it will check the first item in the first list to see if its zero, if it is return true, if not return false.

If the list is not sparse, it just checks if the first item in the list is zero and returns true if it is, false if it is not

**Coeff(x y)**

X is the polynomial you want to check y is the power of x that the coefficient you want to check is connected to. Note it can take in sparse ore dense but will convert both to sparse before calculation, -y should never occur, if it does happen this will loop forever.

First it will check if the list is empty, if it is then return 0

If the list is not empty check if its sparse, if it is sparse then check the second item in the first list, if it equal to y then return the first item of the first list. If y is not equal to the second item in the first list than return coeff((cdr x) y), making it check the next item in the list.

If the list is not sparse than return (coeff (to-sparse x 0) y), which the polynomial to sparse and runs the function again.

**Degree(polynomial)**

Polynomial is the polynomial you want the degree of, note it can take in dense or sparse but will convert to sparse, polynomials with degree 0 will return -inf.0

First it checks if the polynomial is 0, if it is returns -inf.0 if it is not 0 then check if it is sparse. If it is sparse then returns the last item of the last list, this will be the degree.

If it is not sparse then returns (degree (to-sparse polynomial 0)) which converts the polynomial to sparse and runs the function again

**Eval(polynomial x)**

Polynomial is the polynomial you want to evaluate; k is what you want to evaluate it at can take in any list but will convert to sparse before evaluation

First checks if the polynomial is empty, if it is return 0, if it is not empty it will check if its dense if it is dense returns (eval (to-sparse polynomial 0) k), which converts it to sparse and runs the program again.

If the polynomial is sparse then apply an equation to all items in the list, this takes the k value and apply it into our polynomial, then it will run through the equation starting with powers, then multiplication and finally addition, after it returns the int.

**Add-poly(poly1 poly2)**

Poly1 and poly2 are the two polynomials you want to add, only works with dense, used as a helper function for add

Checks 3 conditions, first checks if both lists are empty, if both are returns (), then it will check if poly1 is empty, if it is returns (append (list (car poly2)) ( add-Poly poly1 (cdr poly2))) which is the next item in poly2 and causes recursion,.

Finally if poly2 is empty and poly1 is not it returns (append (list (car poly2)) ( add-Poly poly1 (cdr poly2))) which returns next item in poly1 and triggers recursion

**Add(poly1 poly2)**

Poly1 and poly2 are the polynomials you want to add, works with both dense and sparse but will convert to dense for ease of implementation. It uses is-sparse and add-poly

First it checks if both polynomials are empty, if they are return (0), if not it will check if one is empty, if poly1 is empty it returns poly2, if poly2 is empty it returns poly1.

If no lists are empty it checks the types, it uses is-sparse to check. If both are sparse it returns (to-sparse(add-Poly (to-dense poly1) (to-dense poly2))) which will be the addition applied in dense then converted to sparse before being returned to user

if at least one of the polynomials is not sparse it returns (add-Poly (to-dense poly1) (to-dense poly2))) which will do the addition in dense and return the polynomial in dense form.

**Invert-coef(polynomial)**

Polynomial is the polynomial you want to invert the coefficients of. Only works for dense

Returns (map (lambda (poly1) (\* -1 poly1)) polynomial) which takes every value in the list and multiplies it by -1

**Subtract(poly1 poly2)**

Poly1 will be subtracted by poly2. Works with sparse and dense, but it converts to dense for ease of implementation. Uses is-sparse?, add-poly and invert-coef

First it checks if both lists are sparse, if they are it returns (to-sparse (add-Poly (to-dense poly1) (invert-coef (to-dense poly2)))) which converts both polynomials to dense, inverts the second polynomial, add them together and then converts the result back to sparse before returning the result.

If at least one of the lists is dense than it returns (add-Poly (to-dense poly1) (invert-coef (to-dense poly2))) which converts polynomials to dense, inverts the second polynomial and then adds them.

**Multiply-terms(poly1 poly2)**

Poly1 is the sparse list you want to multiply the first item of. Poly2 is the sparse list that you want the first item of poly1 to multiply into, only works with sparse

First it multiplies the first item of the first list in poly1 by the first item of the first list in poly2, it then adds the second item of the first list in poly1 with the second item of the first list in poly2 effectively retuning a list containing the new coeff and exponent of the first item in the list.

**Apply-poly1-to-p2term(poly1 poly2)**

Poly1 is the poly you want to multiply by, poly2 is the polynomial you want to multiply into. Only works with sparse, uses multiply-terms.

First it checks if poly1 is empty, if it is return() if it is not empty then return (append (multiply-terms (car poly1) poly2) (apply-poly1-to-p2term (cdr poly1) poly2)) this will use multiply term to multiply the first term of each polynomial, then it triggers recursion in order to run through every item in poly1 and multiply into the first item of poly2

**Multiply-poly(poly1 poly2)**

Poly1 is the poly you want to multiply by, poly 2 is the poly you want to multiply poly1 with, only works with sparse uses apply-poly1-p2term poly2 can not be empty.

First checks if length of poly2 is greater than 1, if it is then it will check if it is greater than 2, this is used to make sure the program multiplies every number in the poly without trying to multiply by 0.

So if it is greater than 2 return (append (add (apply-poly1-to-p2term poly1 (car poly2)) (apply-poly1-to-p2term poly1 (cadr poly2))) which uses apply-poly1-to-p2term to multiply every term in poly1 to the first term in poly2, it then triggers recursion.

If length of poly2 is 2 then it will return (append (add (apply-poly1-to-p2term poly1 (car poly2)) (apply-poly1-to-p2term poly1 (cadr poly2)))) which will run apply-poly1-to-p2term on the remaining 2 terms before returning the final result.

If poly2 is 1 than it will return (append(apply-poly1-to-p2term poly1 (car poly2)))

Which runs run apply-poly1-to-p2term on the remaining term before returning the result.

**Multiply(poly1 poly2)**

Poly1 and poly2 are the polynomials you want to multiply together. Works with dense and sparse but converts to dense for ease of implementation, uses multiply-poly is-sparse?.

Checks if either poly is zero, if one of them is returns (0)

Checks if both polynomials are sparse, if they are both sparse returns (multiply-poly poly1 poly2) which multiplies the polynomials together.

If at least one of the polynomials is dense returns (append(apply-poly1-to-p2term poly1 (car poly2))) which converts the polynomials to sparse, multiplies them, then converts them back to dense.

**div-terms(term1 term2)**

term1 is a list corresponding to the first term of a sparse polynomial, term2 is a list corresponding to the first term of the sparse polynomial you want to divide term1 by, note it only works with sparse terms.

It returns (list (list (/ (car term1) (car term2)) (- (cadr term1) (cadr term2)))) which is a list consisting of the first item in term1 divided by the first item in term2 and the second term in term1 minus the second term in term2

**poly-div(poly1 poly2)**

poly1 is the poly you want divided, poly2 is the poly you want to divide by, both must be sparse to work. Uses div-terms

it first checks if the polynomial has already been divided, if it has not already been divided then it returns

(append (div-terms (car poly1) (car poly2)) (poly-div (reverse (subtract (reverse poly1) (multiply (reverse poly2) (div-terms (car poly1) (car poly2))))) poly2))

Which can be simplified to it systematically going through every term and dividing them using div-terms

**quotient(poly1 poly2)**

poly1 is the polynomial you want to find the quotient of, poly2 is the polynomial you want to divide by, works with dense and sparse but will convert to sparse for ease of implementation, uses poly-div.

first it checks if the polynomial is zero, if it is it will return (-inf.0)

if one or both polynomials are dense it returns (to-dense(quotient (to-sparse poly1) (to-sparse poly2))) which converts the polynomials to sparse but will convert them back to dense after it is done the calculations.

If the polynomials are both sparse then return (reverse(poly-div (reverse poly1) (reverse poly2))) which reverses the order to make it work with poly-div runs poly-div then reverses the order before returning the result.

**get-remainder(poly1 poly2)**

poly1 is what you want to divide poly2 is what you are dividing by, uses multiply, divide and subtract. Only works with sparse.

First checks if there is a remainder, does this by using (is-zero? (subtract (multiply (reverse (poly-div (reverse poly1) (reverse poly2))) poly2) poly1)) this does the division on poly1, to find the quotient, multiplies it by poly2 and then checks if subtracting it by the original poly1 = zero, if it does we know that there is no remainder and it will return (0)

If there is a remainder than return (subtract poly1 (multiply (reverse (poly-div (reverse poly1) (reverse poly2))) poly2)) which takes poly1 and subtracts the quotient of poly1 multiplied by poly2 from it, giving us the remainder.

**remainder(poly1 poly2)**

poly1 is the polynomial you want to find the remainder of, poly2 is the polynomial you want to divide by, works with dense and sparse but will convert to sparse for ease of implementation, uses get-remainder.

first it checks if the polynomial is zero, if it is it will return (-inf.0)

if one or both polynomials are dense it returns (to-dense(remainder (to-sparse poly1) (to-sparse poly2))) which converts the polynomials to sparse but will convert them back to dense after it is done the calculations.

If the polynomials are both sparse then return get-remainder(poly1 poly2) which will return the remainder

**power-rule(poly1)**

poly1 is the term in the sparse list you want to apply the power rule to, only works with a sparse term.

First it checks if the second term is less than 1, if it is less than one than return ()

If the first term is one or greater than return list(list (\* (car poly1) (cadr poly1)) (- (cadr poly1) 1)) which is a list with the first term multiplied by the second term and the second term minus 1

**derivative(poly1)**

poly1 is the polynomial you want to find the derivative of, can use sparse and dense but will convert to dense for ease of implementation. Uses power-rule

First checks if polynomial is empty, if it is return ()

Then checks if the polynomial is sparse, if it is sparse returns (append (power-rule (car poly1)) (derivative (cdr poly1))) which uses power-rule to find the derivative of the first term and then triggers recursion to find the next terms derivative

If the list is not sparse, return (to-dense (derivative (to-sparse poly1)) this converts the polynomial to sparse, triggers recursion and after the derivative is calculated converts back to dense.

**normalize-gcd(polynomial coefficient)**

polynomial is the polynomial you want to divide, coefficient is the int you want to divide by, only works with dense.

Returns (map (lambda (term) (/ term coefficent)) Polynomial) which is a list of all terms in polynomial divided by coefficent

**find-gcd(poly1 poly2)**

poly1 and poly2 are the polynomials you want to find the gcd for. Only works with sparse and will convert it to meet the needs of called functions. Uses normalize-gcd, degree, and remainder.

First checks if poly2 is zero if it is we know that we have found gcd, so we normalize it and return it (to-sparse (normalize-gcd (to-dense poly1) (coeff poly1 (degree poly1))))

If poly2 is not 0, the gcd has yet to be found, so continue modifying (find-gcd poly2 (remainder poly1 poly2)) this also triggers recursion to repeat the process

**gcd(poly1 poly2)**

poly1 and poly2 are the polynomials you want to find the gcd of, works with sparse and dense but will convert to sparse for ease of implementation. Uses find-gcd

first check if poly1 and poly2 are zero, if they are both zero return (0)

if only poly1 is zero return poly2

if only poly2 is zero return poly1

if both poly1 and poly2 are sparse then check which one has a lower degree, make it the divisor by returning (find-gcd poly-with-larger-degree poly-with-smaller-degree)

if at least one of the polynomials is dense than return (to-dense (find-gcd (to-sparse poly1) (to-sparse poly2))) to convert it to sparse, do the calculations then convert back to dense before returning