Software Detail Design Document for Symbolic Math Package

Brent Seidel
Phoenix, AZ

March 12, 2009

Contents

Chapter 1

Polynomial Utilities

1.1 Program Headers

{{SWDDD POLY HEADER}}

This file and all items defined in it are contained in the module "Polynomial". Eventually, a data type and class for Polynomial should be added.

1.1.1 Requirements

- The program **shall** {{SWDDD POLY HEADER 10}} be in a module named "Polynomial".
- The functions polyEval, polyAdd, polyMul, polyIntegrate, polyDerive, and polyShow **shall** {{SWDDD POLY HEADER 20}} be exported from the module.

1.1.2 Header Code

```
module Polynomial ( polyEval, polyAdd, polyMul, polyIntegrate, polyDerive, polyShow ) where
```

1.2 Evaluate a Polynomial Using Newton's Method

{{SWRDD POLY EVAL}}

This function evaluates a polynomial at a specific value using Newton's method. A polynomial of standard form:

$$p(X) = a_n X^n + a_{n-1} X^{n-1} \cdots + a_2 X^2 + a_1 X + a_0$$

is rewriten as:

$$p(X) = ((((a_n X) + a_{n-1})X) + \cdots + a_2)X + a_1)X + a_0$$

Evaluating the polynomial in this fashion also minimizes the number of multiplications required.

1.2.1 Requirements

- The newton function **shall** {{SWRDD POLY EVAL 10}} evaluate a polynomial at a specified value.
- The polynomial **shall** {{SWRDD POLY EVAL 20}} be stored in a list containing the a_n coefficients.
- The coefficients in the list **shall** {{SWRDD POLY EVAL 30}} be arranged with a_0 first.

1.2.2 Function Definition

This function evaluates a polynomial using Newton's method.

```
polyEval :: Num a ⇒ [a] → a → a
polyEval [] a = 0
polyEval (x:xs) y = x + y * polyEval xs y
```

1.3 Add Two Polynomials

{{SWRDD POLY ADD}}

This function adds two polynomials of order a_1 and a_2.

$$P_1(X) = \sum_{n=0}^{a_1} c_n X^n$$

and

$$P_2(X) = \sum_{n=0}^{a_2} d_n X^n$$

the sum is given by

$$P_1(X) + P_2(X) = \sum_{n=0}^{MAX(a_1,a_2)} (c_n + d_n) X^n$$

1.3.1 Requirements

- The sum of two polynomials **shall** {{SWRDD POLY ADD 10}} be the sum of the corresponding coefficients.
- Unmatched coefficients **shall** {{SWRDD POLY ADD 20}} be passed unchanged to the output.

1.3.2 Function Definition

This function adds two polynomials producing a third.

```
polyAdd :: (Num a) ⇒ [a] → [a] → [a]
polyAdd [] [] = []
polyAdd x []  = x
polyAdd [] x  = x
polyAdd (x:xs) (y:ys) = x + y :  polyAdd xs ys
```

1.4 Multiply Two Polynomials

{{SWRDD POLY MULTIPLY}}

This function multiplies two polynomials of order a_1 and a_2. Thus:

$$P_1(X) = \sum_{i=0}^{a_1} c_i X^i, P_2(X) = \sum_{j=0}^{a_2} d_j X^j$$

The product then is:

$$P_1(X) \times P_2(X) = \sum_{i=0}^{a_1} c_i X^i \times \sum_{j=0}^{a_2} d_j X^j = \sum_{i=0}^{a_1} \sum_{j=0}^{a_2} c_i d_j X^{i+j}$$

1.4.1 Requirements

- The product of two polynomials **shall** {{SWRDD POLY MULTIPLY 10}} be the sum of the the products of all the monomials.

1.4.2 Function Definition

This function multiplies two polynomials producing a third. This includes a function that multiplies all the coefficients in a polynomial by a scalar value.

```
polyScale :: (Num a) ⇒ [a] → a → [a]
polyScale [] _ = []
polyScale _ 0  = []
polyScale xs y = [x*y | x←xs]

polyMul :: (Num a) ⇒ [a] → [a] → [a]
polyMul [] [] = []
polyMul _ []  = []
polyMul [] _  = []
polyMul (x:xs) y = polyAdd (polyScale y x) (polyMul xs (0 : y))
```

1.5 Integrate a Polynomial

{{SWRDD POLY INTEGRATE}}

The integral of a monomial with respect to X, $c_n X^n$ is given by $\frac{c_n}{n+1} X^{n+1}$. A polynomial is simply a sum of monomials. If

$$P(X) = \sum_{n=0}^{a} c_n X^n$$

then

$$\int P(X)dX = \sum_{n=0}^{a} \int c_n X^n dX = \sum_{n=0}^{a} \frac{c_n}{n+1} X^{n+1}$$

1.5.1 Requirements

- The integral of a polynomial **shall** {{SWRDD POLY INTEGRATE 10}} shall be the sum of the integral of each monomial making up the polynomial.
- The integral of the monomial $c_n X^n$ **shall** {{SWRDD POLY INTEGRATE 20}} be $\frac{c_n}{n+1} X^{n+1}$.
- The constant term in the result **shall** {{SWRDD POLY INTEGRATE 30}} be set to zero.

1.5.2 Function Definition

Note that since division is involved, the coefficients need to be of the fractional type.

```
polyIntegrate :: (Fractional a) ⇒ [a] → [a]
polyIntegrate [] = []
polyIntegrate (x:xs) = 0 : x : helper xs 2
  where
    helper :: (Fractional a) ⇒ [a] → a → [a]
    helper (x:xs) n
      | xs == []  = [x / n]
      | otherwise = (x / n) : helper xs (n + 1)
```

1.6 Find the Derivative of a Polynomial

{{SWRDD POLY DERIVATIVE}}

The derivative of a monomial with respect to X, $c_c X^n$ is given by $nc_n X^{n-1}$. A polynomial is simply a sum of monomials.

1.6.1 Requirements

- The derivative of a polynomial **shall** {{SWRDD POLY DERIVATIVE 10}} shall be the sum of the derivative of each monomial making up the polynomial.
- The derivative of a constant monomial **shall** {{SWRDD POLY DERIVATIVE 20}} be zero.
- The derivative of the monomial $c_n X^n$ **shall** {{SWRDD POLY DERIVATIVE 30}} be $nc_n X^{n-1}$.

1.6.2 Function Definition

```
polyDerive :: (Num a) ⇒ [a] → [a]
polyDerive [] = []
polyDerive (x:xs)
  | xs == []  = [0]
  | otherwise = helper xs 1
  where
    helper :: (Num a) ⇒ [a] → a → [a]
    helper (x:xs) n
      | xs == []  = [n * x]
      | otherwise = (n * x) : helper xs (n + 1)
```

1.7 Display a Polynomial

{{SWRDD POLY DISPLAY}}

This function converts a polynomial in standard form to a string. Polynomials are stored as a list of coefficients.

1.7.1 Requirements

- The coefficients of a polynomials **shall** {{SWRDD POLY DISPLAY 10}} be extracted from a list.
- The coefficients in the list **shall** {{SWRDD POLY DISPLAY 20}} be arranged with the lowest order coefficient first.
- If the coefficient is zero, that term **shall** {{SWRDD POLY DISPLAY 30}} be omitted.
- If the coefficient is one, the coefficient **shall** {{SWRDD POLY DISPLAY 40}} be omitted.

1.7.2 Function Definition

```
polyShow :: (Num a, Ord a) ⇒ [a] → String
polyShow [] = ""
polyShow (x:xs)
  | x == 0        = helper xs 1
  | x > 0         = helper xs 1 ++ " + " ++ show x
```

```
  | x < 0         = helper xs 1 ++ " - " ++ show (-x)
where
  helper :: (Num a, Ord a) ⇒ [a] → Int → String
  helper [] a = ""
  helper (x:xs) n
    | x == 0              = helper xs (n + 1)
    | x == 1 && xs == []  = helper xs (n + 1) ++ "X^" ++ show n
    | x > 0 && xs ==[]    = helper xs (n + 1) ++  show x ++ "X^" ++ show n
    | x < 0 && xs == []   = helper xs (n + 1) ++ show x ++ "X^" ++ show n
    | x == 1              = helper xs (n + 1) ++ " + " ++ "X^" ++ show n
    | x > 0               = helper xs (n + 1) ++ " + " ++ show x ++ "X^" ++ show n
    | x < 0               = helper xs (n + 1) ++ " - " ++ show (-x) ++ "X^" ++ show n
```

Chapter 2

Parsing Utilities

2.1 Program Headers

{{SWDDD PARSE HEADER}}

This file and all items defined in it are contained in the module "Parse".

2.1.1 Requirements

- The program **shall** {{SWDDD PARSE HEADER 10}} be in a module named "Parse".
- The Char module **shall** {{SWDDD PARSE HEADER 20}} be imported to provide character functions.
- Only the parsePoly function **shall** {{SWDDD PARSE HEADER 30}} be exported.

2.1.2 Header Code

```
module Parse (parsePoly) where

import Char (isDigit, isSpace)
```

2.2 Get Coefficient

{{SWDDD PARSE POLY COEF}}

This function is part of the tokenizer and is used to extract a coefficient from the input string. The coefficient should be a valid double precision number.

2.2.1 Requirments

- A coefficient **shall** {{SWDDD PARSE POLY COEF 10}} be a double precision number.
- If a coefficient is omitted, its value **shall** {{SWDDD PARSE POLY COEF 20}} be set to 1.
- If the sign is omitted, it **shall** {{SWDDD PARSE POLY COEF 30}} be assumed to be positive.
- Whitespace characters **shall** {{SWDDD PARSE POLY COEF 40}} be ignored.
- The digits before or after the decimal point **shall** {{SWDDD PARSE POLY COEF 50}} be optional.
- The decimal point **shall** {{SWDDD PARSE POLY COEF 60}} be optional.
- The exponent **shall** {{SWDDD PARSE POLY COEF 70}} be optional.
- If the exponent is present, it **shall** {{SWDDD PARSE POLY COEF 80}} consist of the letter 'E' followed by an optional sign and non-optional digits.
- The coefficient **shall** {{SWDDD PARSE POLY COEF 90}} be terminated by the end of the input string, a sign, the letter 'X', or an error condition.

2.2.2 Function Definitions

The coefficient parser is implemented using a state machine. Some helper functions are also defined.

Define Support

Define a data type for the states and some helper functions.

```
data GetCoefStates = StCoefStart | StCoefFoundSign | StCoefPreDecimal | StCoefPostDecimal |
StCoefFoundExp | StCoefExpSign | StCoefExpValue

isSign :: Char → Bool
isSign x = (x == '+') || (x == '-')

isExp :: Char → Bool
isExp x = (x == 'e') || (x == 'E')

isX :: Char → Bool
isX x = (x == 'x') || (x == 'X')
```

Define the Main Coefficient Parser

The parser takes a state and a string as inputs and returns a tuple consisting of two strings. The first string is the parsed data and the second string is the remaining data.

If no input string is passed, two null strings are returned.

```
getCoefficient :: GetCoefStates → String → (String, String)
getCoefficient _ [] = ([], [])
```

The Start State

In the first state, we can take a digit, a sign, a decimal point, or an "X".

```
getCoefficient StCoefStart (x:xs)
  | isSpace x = (skip1, skip2)
  | isDigit x = (x : dig1, dig2)
  | x == '-'  = (x : sign1, sign2)
  | x == '+'  = (sign1, sign2)
  | isX x     = ("1", x : xs)
  | x == '.'  = (x : dig3, dig4)
  | otherwise = ("*Error", "*Error")
    where
      (skip1, skip2) = getCoefficient StCoefStart xs
      (dig1, dig2) = getCoefficient StCoefPreDecimal xs
      (dig3, dig4) = getCoefficient StCoefPostDecimal xs
      (sign1, sign2) = getCoefficient StCoefFoundSign xs
```

The Found Sign State

Once a sign has been found, signs are no longer valid characters.

```
getCoefficient StCoefFoundSign (x:xs)
  | isSpace x = (skip1, skip2)
  | isDigit x = (x : dig1, dig2)
  | x == '.'  = (x : dig3, dig4)
  | isX x     = ("1", x : xs)
  | otherwise = ("*Error", "*Error")
    where
      (skip1, skip2) = getCoefficient StCoefFoundSign xs
      (dig1, dig2) = getCoefficient StCoefPreDecimal xs
      (dig3, dig4) = getCoefficient StCoefPostDecimal xs
```

The Pre-Decimal Point State

Now, process the digits before the decimal point.

```
getCoefficient StCoefPreDecimal (x:xs)
  | isSpace x = (dig1, dig2)
  | isDigit x = (x : dig1, dig2)
  | isExp x   = (x : exp1, exp2)
  | isX x     = ([], x : xs)
  | isSign x  = ([], x : xs)
  | x == '.'  = (x : dec1, dec2)
  | otherwise = ("*Error", "*Error")
    where
      (dig1, dig2) = getCoefficient StCoefPreDecimal xs
      (dec1, dec2) = getCoefficient StCoefPostDecimal xs
      (exp1, exp2) = getCoefficient StCoefFoundExp xs
```

The Post-Decimal Point State

If a decimal point has been found, process any digits after it.

```
getCoefficient StCoefPostDecimal (x:xs)
  | isSpace x = (dig1, dig2)
  | isDigit x = (x : dig1, dig2)
  | isExp x   = (x : exp1, exp2)
  | isX x     = ([], x : xs)
  | isSign x  = ([], x : xs)
  | otherwise = ("*Error", "*Error")
    where
      (dig1, dig2) = getCoefficient StCoefPostDecimal xs
      (exp1, exp2) = getCoefficient StCoefFoundExp xs
```

The Found Exponent State

If an "E" has been found, look for the exponent.

```
getCoefficient StCoefFoundExp (x:xs)
  | isSpace x = (exp1, exp2)
  | isDigit x = (x : dig1, dig2)
  | isSign x  = (x : sign1, sign2)
  | otherwise = ("*Error", "*Error")
    where
      (exp1, exp2) = getCoefficient StCoefFoundExp xs
      (sign1, sign2) = getCoefficient StCoefExpSign xs
      (dig1, dig2) = getCoefficient StCoefExpValue xs
```

The Exponent Sign State

The exponent should have a sign.

```
getCoefficient StCoefExpSign (x:xs)
  | isSpace x = (sign1, sign2)
  | isDigit x = (x : dig1, dig2)
  | otherwise = ("*Error", "*Error")
    where
      (sign1, sign2) = getCoefficient StCoefExpSign xs
      (dig1, dig2) = getCoefficient StCoefExpValue xs
```

The Exponent Value State

The exponent value should consist only of digits.

```
getCoefficient StCoefExpValue (x:xs)
  | isSpace x = (dig1, dig2)
  | isDigit x = (x : dig1, dig2)
```

```
  | isSign x  = ([], x : xs)
  | isX x     = ([], x : xs)
  | otherwise = ("*Error", "*Error")
    where
      (dig1, dig2) = getCoefficient StCoefExpValue xs
```

2.3 Get Exponent

{{SWDDD PARSE POLY EXP}}

This function is part of the tokenizer and is used to extract an exponent from the input string. The exponent should be a positive integer.

2.3.1 Requirments

- An exponent **shall** {{SWDDD PARSE POLY EXP 10}} be a positive integer.
- If an exponent is omitted and the 'X' is present, its value **shall** {{SWDDD PARSE POLY EXP 20}} be set to 1.
- If both the exponent and the 'X' are omitted, its value **shall** {{SWDDD PARSE POLY EXP 30}} be set to 0.
- Whitespace characters **shall** {{SWDDD PARSE POLY EXP 40}} be ignored.
- The exponent **shall** {{SWDDD PARSE POLY EXP 90}} be terminated by the end of the input string, a sign, or an error condition.

2.3.2 Function Definitions

The exponent parser is similar to the coefficient parser, but simpler. All it needs to do is to look for the independent variable, "X", the exponent symbol, and then a string of digits.

Define Support

A data type for the states is defined.

```
data GetExpStates = StExpStart | StExpFoundX | StExpFoundPow | StExpDigits
```

Define the Exponent Parser

The exponent parser definition is similar to the coefficient parser. It takes a state and a string and returns two string. If, in the start state, the input string is null, an exponent of zero is returned. If the input string is null, two null strings are returned.

```
getExponent :: GetExpStates → String → (String, String)

getExponent StExpStart [] = ("0", [])
getExponent StExpFoundX [] = ("1", [])
getExponent _ [] = ([], [])
```

The Start State

In this state, an "X" is expected. If a sign is found, then there is no independent variable and the exponent is zero.

```
getExponent StExpStart (x:xs)
  | isSpace x = (skip1, skip2)
  | isSign x  = ("0", x : xs)
  | isX x     = (dig1, dig2)
  | otherwise = ("*Error", "*Error")
    where
      (skip1, skip2) = getExponent StExpStart xs
      (dig1, dig2) = getExponent StExpFoundX xs
```

The Found X State

In this state, we should either find a sign or an exponent symbol.

```
getExponent StExpFoundX (x:xs)
  | isSpace x    = (skip1, skip2)
  | x == '^'     = (pow1, pow2)
  | isSign x     = ("1", x : xs)
  | otherwise    = ("*Error", "*Error")
    where
      (skip1, skip2) = getExponent StExpFoundX xs
      (pow1, pow2) = getExponent StExpFoundPow xs
```

The Found Power State

Once the exponent symbol has been found, the only valid thing to follow is a digit.

```
getExponent StExpFoundPow (x:xs)
  | isSpace x = (skip1, skip2)
  | isDigit x = (x : dig1, dig2)
  | otherwise = ("*Error", "*Error")
    where
```

```
      (dig1, dig2) = getExponent StExpDigits xs
      (skip1, skip2) = getExponent StExpFoundPow xs
```

The Exponent Digits State

Now, just accumulate all the digits until something else is found.

```
getExponent StExpDigits (x:xs)
  | isSpace x = (dig1, dig2)
  | isDigit x = (x : dig1, dig2)
  | isSign x  = ([], x : xs)
  | otherwise = ("*Error", "*Error")
    where
      (dig1, dig2) = getExponent StExpDigits xs
```

2.4 Tokenizer

{{SWDDD PARSE POLY TOKEN}}

The tokenizer reads through the input string and breaks it up into tokens.

2.4.1 Requirments

- A parsed polynomial string **shall** {{SWDDD PARSE POLY TOKEN 10}} return a list of coefficients and exponents.
- The coefficients **shall** {{SWDDD PARSE POLY TOKEN 20}} be double precision floating point numbers.
- The exponents **shall** {{SWDDD PARSE POLY TOKEN 30}} be integers.
- The independent variable **shall** {{SWDDD PARSE POLY TOKEN 40}} be 'X'.
- If the exponent on 'X' is omitted, it **shall** {{SWDDD PARSE POLY TOKEN 50}} be assumed to be 1.
- If the coefficient is not followed by the independent variable, the exponent **shall** {{SWDDD PARSE POLY TOKEN 60}} be assumed to be 0.

2.4.2 Function Definitions

```
tokenize :: String → [(Double, Int)]
tokenize [] = []
tokenize x
  | rem1 == "*Error" = []
  | rem2 == "*Error" = []
  | otherwise        = (coeff, exp) : tokenize rem2
  where
```

```
  (coeff1, rem1) = getCoefficient StCoefStart x
  (exp1, rem2) = getExponent StExpStart rem1
  coeff = (read coeff1) :: Double
  exp = (read exp1) :: Int
```

2.5 Build

{{SWDDD PARSE POLY BUILD}}

The build process takes a tokenized polynomial (a list of coefficients and exponents) and creates a polynomial array.

2.5.1 Requirments

- The build process **shall** {{SWDDD PARSE POLY BUILD 10}} take the tokenized polynomial and construct a polynomial array.
- The polynomial array **shall** {{SWDDD PARSE POLY BUILD 20}} be an array of double precision numbers.
- The polynomial array **shall** {{SWDDD PARSE POLY BUILD 30}} contain the coefficients.
- The coefficients **shall** {{SWDDD PARSE POLY BUILD 40}} be located in the indices corresponding to the exponents.

2.5.2 Function Definitions

```
maxToken :: Int → [(Double, Int)] → Int
maxToken x [] = x

maxToken x (y:ys)
  | x > t = maxToken x ys
  | otherwise = maxToken t ys
  where
    (_, t) = y

zeros :: [Double]
zeros = 0.0 : zeros

updatePoly :: [(Double, Int)] → [Double] → [Double]
updatePoly [] x = x

updatePoly (x:xs) y = result
  where
    (coeff, index) = x
    beg = take index y
    end = drop (index + 1) y
```

```
    mid = y!!index + coeff
    result = updatePoly xs (beg ++ [mid] ++ end)

buildPoly :: [(Double, Int)] → [Double]
buildPoly [] = []

buildPoly x = result
  where
    order = maxToken 0 x
    arry = take (order + 1) zeros
    result = updatePoly x arry
```

2.6 Parse

{{SWDDD PARSE POLY PARSE}}

The polynomial parser combines the process of the tokenizer and the builder.

2.6.1 Requirments

- The parser **shall** {{SWDDD PARSE POLY PARSE 10}} convert a string containing a polynomial into an array of double precision numbers.

2.6.2 Function Definitions

```
parsePoly :: String → [Double]
parsePoly [] = []

parsePoly x = buildPoly (tokenize x)
```

Chapter 3

Zero Finding Methods

3.1 Program Headers

{{SWDDD ZERO HEADER}}

This file and all items defined in it are contained in the module "Solver". This module contains routines to solve equations. In most cases this involves finding "zeros" of a function.

3.1.1 Requirements

- The program **shall** {{SWDDD ZERO HEADER 10}} be in a module named "Solver".

3.1.2 Header Code

```
module Solver where
```

3.2 Bisection Method

{{SWRDD ZERO BISECT}}

The bisection method starts with two guesses for the zero. The function must evaluate to a positive value at one guess and a negative value at the other. At each iteration, the middle value is tested to determine on which side the zero lies. The interval is then reduced by half. After a sufficient number of iterations or a sufficiently small interval is reached, the value is returned.

Note that each iteration produces one additional bit of the solution, so convergence is linear.

3.2.1 Requirements

- The bisection algorithm **shall** {{SWRDD ZERO BISECT 10}} accept as input a function, a lower limit, and an upper limit.
- The bisection algorithm **shall** {{SWRDD ZERO BISECT 20}} return a value for the zero and a string containing a message.
- The input function **shall** {{SWRDD ZERO BISECT 30}} accept a numeric input and return a numeric result.
- The value of the input function **shall** {{SWRDD ZERO BISECT 40}} be checked at the upper and lower limit to ensure that the signs are opposite.
- If the signs are not opposite, an error message **shall** {{SWRDD ZERO BISECT 50}} be returned.
- The iteration limit **shall** {{SWRDD ZERO BISECT 60}} be set to 64.
- If a zero is reached exactly, iteration **shall** {{SWRDD ZERO BISECT 70}} be terminated and that value returned.

3.2.2 Function Definition

This function finds a zero of a function using the bisection method.

```
bisect :: (Floating a, Ord a) ⇒ (a → a) → a → a → (a, String)
bisect f lower upper
  | (f lower) ∗ (f upper) ≥ 0  = (0, "ERROR: Function has same sign at upper and lower limits.")
  | otherwise                  = bisectHelper f lower upper 64
  where
    bisectHelper :: (Floating a, Ord a) ⇒ (a → a) → a → a → Int → (a, String)
    bisectHelper f lower upper limit
      | limit ≤ 0              = (mid, "Zero found between " ++ show lower ++ " and " ++ show upper)
      | (f mid) == 0           = (mid, "Zero found exactly with " ++ show limit ++ " iterations left to go.")
      | (f lower) ∗ (f mid) < 0 = bisectHelper f lower mid (limit - 1)
      | otherwise              = bisectHelper f mid upper (limit - 1)
      where
        mid = (lower + upper) / 2
```

3.3 Newton's Method

{{SWRDD ZERO NEWTON}}

Newton's method (aka Newton-Raphson method) is one of the better known root finding algorithms. Its main drawback is that it depends on having the derivative of the function. The basic iteration step is:

$$p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)}$$

Note that if $f'(p_n)$ becomes zero, the iteration cannot be continued.

When it works, it can achieve quadratic convergence. However, it is not always guaranteed to work.

3.3.1 Requirements

- The Newton's method algorithm **shall** {{SWRDD ZERO NEWTON 10}} accept a function, its derivative, a starting guess, a tolerance, and number of iterations.
- The Newton's method algorithm **shall** {{SWRDD ZERO NEWTON 20}} return a result and a message indicating the meaning of the result.
- If the derivative of the function is equal to zero, iteration **shall** {{SWRDD ZERO NEWTON 30}} be terminated and an error message returned.
- If the required tolerance is not reached before the allowed number of itereations, an error message **shall** {{SWRDD ZERO NEWTON 40}} be returned.

3.3.2 Function Definition

```
newton :: (Floating a, Ord a) ⇒ (a → a) → (a → a) → a → a → Int →(a, String)
newton f fP guess tol limit
  | fP guess == 0           = (guess, "ERROR: Derivative of function reached zero.  Cannot continue.")
  | abs(next - guess) < tol = (guess, "Solution found with " ++ show limit ++ " iterations remaining.")
  | limit ≤ 0               = (guess, "ERROR: No solution within tolerance within allowed iterations.")
  | otherwise               = newton f fP next tol (limit - 1)
  where
    next = guess - (f guess) / (fP guess)
```

3.4 Müler's Method

{{SWRDD ZERO MÜLER}}

3.4.1 Requirements

3.4.2 Function Definition

Chapter 4

Main Program

4.1 Program Headers

{{SWDDD MAIN HEADER}}

The main part of the program is contained in the "Main" module.

4.1.1 Requirements

- The program **shall** {{SWDDD MAIN HEADER 10}} be in a module named "Main".
- The Main module **shall** {{SWDDD MAIN HEADER 20}} import the Polynomial module.
- The Main module **shall** {{SWDDD MAIN HEADER 30}} import the Parse module.
- The Main module **shall** {{SWDDD MAIN HEADER 40}} import the Solver module.
- Polynomials **shall** {{SWDDD MAIN HEADER 50}} be stored in a list indexed by polynomial name.
- The polynomial name **shall** {{SWDDD MAIN HEADER 60}} be a String.
- The polynomial **shall** {{SWDDD MAIN HEADER 70}} be a list of double.

4.1.2 Header Code

```
module Main where

import Char (isDigit, isSpace, isLower, toUpper)
import Text.Printf
import Polynomial
import Parse
import Solver
```

```
import Debug.Trace

type Name = String
type Poly = [Double]
type Entry = (String, [Double])
```

4.2 Parse Utilities

{{SWDDD MAIN UTIL}}

4.2.1 Requirements

- The first word function **shall** {{SWDDD MAIN UTIL 10}} return the first word of a line and the remainder of the line.
- The uppercase function **shall** {{SWDDD MAIN UTIL 20}} check if each character is lowercase, and if so, convert it to uppercase.
- The find polynomial function **shall** {{SWDDD MAIN UTIL 30}} search the polynomial library for a specified polynomial and return it.
- If no the specified polynomial is not found, the find polynomial function **shall** {{SWDDD MAIN UTIL 40}} return an empty list.

4.2.2 Function Definition

Function to convert a string to uppercase.

```
upCase :: String → String
upCase [] = []
upCase (x:xs)
  | isLower x = (toUpper x) : upCase xs
  | otherwise = x : upCase xs
```

Function to return the first word of a string, where words are separated by white space.

```
firstWord :: String → (String, String)
firstWord [] = ([], [])
firstWord (x:xs)
  | isSpace x = ([], xs)
  | otherwise = (x : cmd, rest)
    where
      (cmd, rest) = firstWord xs
```

Function to search the polynomial library for the name of a polynomial

```
findPoly :: [(String, [Double])] → String → [Double]
findPoly [] _ = []

findPoly (x:xs) key
  | key == name = poly
  | otherwise   = findPoly xs key
  where
    (name, poly) = x
```

4.3 Command Dispatcher

{{SWDDD MAIN CMD}}

4.3.1 Requirements

- The command dispatcher **shall** {{SWDDD MAIN CMD 10}} call the appropriate function for each command.
- If an unrecognized command is given, an error message **shall** {{SWDDD MAIN CMD 20}} be produced.

4.3.2 Function Definition

```
dispatch :: [Entry] → String → String → ([Entry], String)
dispatch lib cmd rest
  | cmd == "ADD"       = binPoly lib rest polyAdd
  | cmd == "DEL"       = delPoly lib rest
  | cmd == "DERIVE"    = calcPoly lib rest polyDerive
  | cmd == "DIV"       = (lib, "Polynomial division not yet implemented.")
  | cmd == "EVAL"      = evaluate lib rest
  | cmd == "EXIT"      = (lib, "Exit command entered.")
  | cmd == "FACTOR"    = (lib, "Polynomial factoring not yet implemented.")
  | cmd == "HELP"      = (lib, help)
  | cmd == "INTEGRATE" = calcPoly lib rest polyIntegrate
  | cmd == "LIST"      = list lib
  | cmd == "MUL"       = binPoly lib rest polyMul
  | cmd == "NEW"       = newPoly lib rest
  | cmd == "QUIT"      = (lib, "Quit command entered.")
  | cmd == "ROOT"      = root lib rest
  | cmd == "TABLE"     = table lib rest
  | otherwise          = (lib, "<" ++ cmd ++ "> is not a valid command.")
```

4.4 Help Message

{{SWDDD MAIN HELP}}

4.4.1 Requirements

- The help message **shall** {{SWDDD MAIN HELP 10}} contain a help message for the program.

- The help message **shall** {{SWDDD MAIN HELP 20}} identify the valid commands, parameters, and function of the command.

4.4.2 Function Definition

```
help :: String
help = "The following commands are defined.λn" ++
       "ADD P1 P2 P3λn" ++
       "   (add polynomials P1 and P2 together producing P3)λn" ++
       "DEL P1λn" ++
       "   (deletes polynomial named P1 from the saved polynomials)λn" ++
       "DERIVE P1 P2λn" ++
       "   (calculate the derivative of polynomial P1 and save as P2)λn" ++
       "DIVλn" ++
       "   (command not yet implemented)λn" ++
       "EVAL P1 VALUEλn" ++
       "   (calculate the value of polynomial P1 at VALUE)λn" ++
       "EXITλn" ++
       "   (exit the program)λn" ++
       "FACTORλn" ++
       "   (command not yet implemented)λn" ++
       "HELPλn" ++
       "   (produce this help message)λn" ++
       "INTEGRATE P1 P2λn" ++
       "   (calculate the integral of polynomial P1 and save as P2)λn" ++
       "LISTλn" ++
       "   (print a list of all saved polynomials)λn" ++
       "MUL P1 P2 P3λn" ++
       "   (multiply polynomials P1 and P2 together producing P3)λn" ++
       "NEW NAME POLYλn" ++
       "   (parse POLY as a polynomial and save it with the name NAME)λn" ++
       "QUITλn" ++
       "   (quit the program)λn" ++
       "ROOT TYPE P1 OTHERλn" ++
       "   (find a root of P1 using method TYPE.  OTHER depends on type selectedλn" ++
       "ROOT BISECT P1 LOWER UPPERλn" ++
       "   (use bisection method with LOWER and UPPER limits)λn" ++
       "ROOT NEWTON P1 GUESS TOL ITERλn" ++
       "   (use Newton's method with starting GUESS, tolerance TOL and iterationλn" ++
       "    limit ITER)λn" ++
       "TABLE P1 START STOP INCRλn" ++
       "   (print a table of values for P1 starting from START an continuing toλn" ++
       "    STOP with increment INCR)λn"
```

4.5 Command Loop

{{SWDDD MAIN LOOP}}

4.5.1 Requirements

- The main loop **shall** {{SWDDD MAIN LOOP 10}} print a prompt and request a line of input.
- If the EXIT command is given, the main loop **shall** {{SWDDD MAIN LOOP 20}} terminate.
- All other commands **shall** {{SWDDD MAIN LOOP 30}} be passed to the command dispatcher function

4.5.2 Function Definition

```
mainLoop :: [Entry] → IO ()

mainLoop lib = do
  putStr "> "
  line ← getLine
  let (cmd, rest) = firstWord (upCase line)
      (newLib, msg) = dispatch lib cmd rest
  putStrLn msg
  if cmd == "EXIT" || cmd == "QUIT"
  then return ()
  else mainLoop newLib
```

4.6 List Command

{{SWDDD MAIN LIST}}

4.6.1 Requirements

- The list command **shall** {{SWDDD MAIN LIST 10}} list all the saved polynomials.
- If no polynomials have been saved, an appropriate message **shall** {{SWDDD MAIN LIST 20}} be displayed.

4.6.2 Function Definition

Two functions are defined. The list function checks if the library is null and prints an appropriate message. If the library is not null, the temp list iterates through it and prints out the contents.

```
list :: [Entry] → ([Entry], String)
list [] = ([], "No polynomials have been saved.")

list lib = (lib, tempList lib)

tempList :: [Entry] → String
tempList [] = ""
tempList (x:xs) = "<" ++ name ++ "> = <" ++ (polyShow poly) ++ ">λn" ++ tempList xs
  where (name, poly) = x
```

4.7 New Polynomial Command

{{SWDDD MAIN NEW}}

4.7.1 Requirements

- The new command **shall** {{SWDDD MAIN NEW 10}} parse a polynomial and add it to the list.
- Each polynomial added to the list **shall** {{SWDDD MAIN NEW 20}} have a unique name associated with it.
- If the specified name is not unique, an error message **shall** {{SWDDD MAIN NEW 30}} be displayed and no new entry made.

4.7.2 Function Definition

The new polynomial function inserts a new polynomial into the library. It defines three functions that are used locally. Note that new entries are added to the beginning of the library list.

```
newPoly :: [Entry] → String → ([Entry], String)
newPoly lib [] = (lib, "No polynomial or name specified")

newPoly lib rest = checkName lib name polyText
  where
    (name, polyText) = firstWord rest

    checkName :: [Entry] → String → String → ([Entry], String)
    checkName lib name polyText
      | name == ""              = (lib, "No name specified for polynomial")
      | findPoly lib name ≠ [] = (lib, "Name <" ++ name ++ "> already exists")
      | otherwise               = checkPoly lib name polyText

    checkPoly ::  [Entry] → String → String → ([Entry], String)
    checkPoly lib name polyText = insertPoly lib name polyList
      where
        polyList = parsePoly polyText
```

```
insertPoly :: [Entry] → String → Poly → ([Entry], String)
insertPoly lib _ [] = (lib, "No valid polynomial specified.")
insertPoly lib "" _ = (lib, "No polynomial name specified.")
insertPoly lib name polyList = ((name, polyList) : lib,
  "Polynomial added to library.")
```

4.8 Delete Polynomial Command

{{SWDDD MAIN DEL}}

4.8.1 Requirements

- The delete command **shall** {{SWDDD MAIN DEL 10}} remove the specified polynomial from the library.

4.8.2 Function Definition

```
delPoly :: [Entry] → String → ([Entry], String)
delPoly lib [] = (lib, "No polynomial name specified.")

delPoly lib name
  | findPoly lib name == [] = (lib, "Polynomial <" ++ name ++ "> does not exists")
  | otherwise               = ((removePoly lib name), "Polynomial <" ++ name ++ "> deleted.")
  where
    removePoly ::  [Entry] → String → [Entry]
    removePoly [] name = []
    removePoly (l:ib) index
      | index == name = removePoly ib index
      | otherwise     = l : (removePoly ib index)
      where
        (name, _) = l
```

4.9 Binary Polynomial Commands

{{SWDDD MAIN BINARY}}

4.9.1 Requirements

- The binary polynomial commands **shall** {{SWDDD MAIN BINARY 10}} take two polynomials for input and produce a single polynomial output.

- The input polynomials **shall** {{SWDDD MAIN BINARY 20}} be specified by name in the polynomial library.
- The output polynomial **shall** {{SWDDD MAIN BINARY 30}} be specified as a name to be added to the polynomial library.

4.9.2 Function Definition

The type signature for the checkArg helper function is commented out because I couldn't find one that would work.

```
binPoly :: [Entry] → String → ([Double] → [Double] → [Double]) → ([Entry], String)
binPoly lib [] _ = (lib, "No polynomials specified.")

binPoly lib rest op = (newLib, msg)
  where
    (p1, rest1) = firstWord rest
    (p2, p3) = firstWord rest1
    (newLib, msg) = checkArg lib p1 p2 p3 op

--    checkArg :: [Entry] → String → String → String → ([Double] → [Double] → [Double]) →
([Entry], String)
    checkArg lib "" _ _ _ = (lib, "No input polynomial specified.")
    checkArg lib _ "" _ _ = (lib, "No input polynomial specified.")
    checkArg lib _ _ "" _ = (lib, "No output polynomial specified.")
    checkArg lib p1 p2 p3 op
      | poly1 == [] = (lib, "Polynomial <" ++ p1 ++ "> not found.")
      | poly2 == [] = (lib, "Polynomial <" ++ p2 ++ "> not found.")
      | poly3 ≠ [] = (lib, "Polynomial <" ++ p3 ++ "> already exists.")
      | otherwise   = ((p3, (op poly1 poly2)) : lib, "New polynomial created.")
      where
        poly1 = findPoly lib p1
        poly2 = findPoly lib p2
        poly3 = findPoly lib p3
```

4.10 Calculus Polynomial Commands

{{SWDDD MAIN CALC}}

4.10.1 Requirements

- The calculus polynomial commands **shall** {{SWDDD MAIN CALC 10}} take one polynomial for input and produce a single polynomial output.
- The input polynomial **shall** {{SWDDD MAIN CALC 20}} be specified by name in the polynomial library.

- The output polynomial **shall** {{SWDDD MAIN CALC 30}} be specified as a name to be added to the polynomial library.

4.10.2 Function Definition

The type signature for the checkArg helper function is commented out because I couldn't find one that would work.

```
calcPoly :: [Entry] → String → ([Double] → [Double]) → ([Entry], String)
calcPoly lib [] _ = (lib, "No polynomials specified.")

calcPoly lib rest op = (newLib, msg)
  where
    (p1, p2) = firstWord rest
    (newLib, msg) = checkArg lib p1 p2 op

--    checkArg :: [Entry] → String → String → ([Double] → [Double] → [Double]) → ([Entry], String)
    checkArg lib "" _ _ = (lib, "No input polynomial specified.")
    checkArg lib _ "" _ = (lib, "No output polynomial specified.")
    checkArg lib p1 p2 op
      | poly1 == [] = (lib, "Polynomial <" ++ p1 ++ "> not found.")
      | poly2 ≠ [] = (lib, "Polynomial <" ++ p2 ++ "> already exists.")
      | otherwise   = ((p2, op poly1) : lib, "New polynomial created.")
      where
        poly1 = findPoly lib p1
        poly2 = findPoly lib p2
```

4.11 Evaluate Polynomial Command

{{SWDDD MAIN EVAL}}

4.11.1 Requirements

- The evaluate polynomial command **shall** {{SWDDD MAIN EVAL 10}} take one polynomial and a number for input and produce a single number output.
- The input polynomial **shall** {{SWDDD MAIN EVAL 20}} be specified by name in the polynomial library.
- The specified polynomial **shall** {{SWDDD MAIN EVAL 30}} be evaluated at the specified value.

4.11.2 Function Definition

```
evaluate :: [Entry] → String → ([Entry], String)
evaluate lib [] = (lib, "No polynomials specified.")

evaluate lib rest = (newLib, msg)
  where
    (p1, value) = firstWord rest
    (newLib, msg) = checkArg lib p1 ((read value) :: Double)

    checkArg :: [Entry] → String → Double → ([Entry], String)
    checkArg lib "" _ = (lib, "No input polynomial specified.")
    checkArg lib p1 value
      | poly1 == [] = (lib, "Polynomial <" ++ p1 ++ "> not found.")
      | otherwise   = (lib, msg)
      where
        poly1 = findPoly lib p1
        msg = "Value is " ++ (show (polyEval poly1 value))
```

4.12 Table Command

{{SWDDD MAIN TABLE}}

4.12.1 Requirements

- The table commands **shall** {{SWDDD MAIN TABLE 10}} take one polynomial, a starting value, an ending value, and an increment and produce a table.
- The input polynomial **shall** {{SWDDD MAIN TABLE 20}} be specified by name in the polynomial library.
- The specified polynomial **shall** {{SWDDD MAIN TABLE 30}} be evaluated at each value starting at the starting value, incrementing by the increment amount, until the ending value is reached.

4.12.2 Function Definition

The table function consists of a main function and three helper functions.

```
table :: [Entry] → String → ([Entry], String)
table lib [] = (lib, "No polynomials specified.")

table lib rest = (newLib, msg)
  where
    (p1, r1) = firstWord rest
    (start, r2) = firstWord r1
    (end, incr) = firstWord r2
    (newLib, msg) = checkArg lib p1 ((read start) :: Double) ((read end) :: Double) ((read incr) :: Double)

    checkArg :: [Entry] → String → Double → Double → Double → ([Entry], String)
```

```
checkArg lib "" _ _ _ = (lib, "No input polynomial specified.")
checkArg lib p1 start end incr
  | poly1 == [] = (lib, "Polynomial <" ++ p1 ++ "> not found.")
  | otherwise   = (lib, msg)
  where
    poly1 = findPoly lib p1
    msg = "  X  P(X)\n" ++ formatTable (tableHelp poly1 start end incr)

tableHelp :: Poly → Double → Double → Double → [(Double, Double)]
tableHelp [] _ _ _ = []
tableHelp _ _ _ 0 = []
tableHelp p1 start end incr
  | start > end = []
  | incr < 0    = []
  | otherwise   = (start, (polyEval p1 start)) : tableHelp p1 (start + incr) end incr

formatTable :: [(Double, Double)] → String
formatTable [] = ""
formatTable (x:xs) = (printf "%g  %g\n" y fy) ++ formatTable xs
  where
    (y, fy) = x
```

4.13 Root Command

{{SWDDD MAIN ROOTS}}

This function uses the root finding functions from the Solver module to find zeros of a polynomial. Note that currently, only real roots are supported.

4.13.1 Requirements

- The root finding function **shall** {{SWDDD MAIN ROOTS 10}} take a method, a polynomial, and one, or more initial guesses.
- The bisection method **shall** {{SWDDD MAIN ROOTS 20}} take two initial guesses that must bracket the root.
- The newton's method **shall** {{SWDDD MAIN ROOTS 30}} take one initial guess.

4.13.2 Function Definition

The root command determines which root finding algorithm has been specified and calls the appropriate algorithm.

```
root :: [Entry] → String → ([Entry], String)
root lib [] = (lib, "No method or polynomial specified.")

root lib rest = (lib, msg)
  where
    (method, r1) = firstWord rest
    (p1, r2) = firstWord r1
    poly = findPoly lib p1
    msg = findMethod method poly r2
```

Determine which root finding method has been requested.

```
findMethod :: String → Poly → String → String
findMethod method poly rest
  | method == "BISECT" = findBisect poly rest
  | method == "NEWTON" = findNewton poly rest
  | poly == []         = "Polynomial not found in library."
  | otherwise          = "Method <" ++ method ++ "> is not a defined root finding method."
```

Wrapper for the bisection algorithm. Parse out the upper and lower limits and call the bisection algorithm.

```
findBisect :: Poly → String → String
findBisect poly rest
  | l == "" = "No lower limit specified."
  | u == "" = "No upper limit specified."
  | lower ≥ upper = "Lower limit must be less than upper limit."
  | otherwise = msg
  where
    (l, u) = firstWord rest
    lower = (read l) :: Double
    upper = (read u) :: Double
    f = polyEval poly
    (value, txt) = bisect f lower upper
    msg = "Zero value of " ++ (show value) ++ " found.λn" ++ txt
```

Wrapper for Newton's method algorithm. Parse out the guess, tolerance, and iteration limit. Determine derivative of the polynomial. Call Newton's method.

```
findNewton :: Poly → String → String
findNewton poly rest
  | g == "" = "No guess specified."
  | t == "" = "No tolerance specified."
  | l == "" = "No iteration limit specified."
  | otherwise = msg
  where
    (g, r) = firstWord rest
    (t, l) = firstWord r
    guess = (read g) :: Double
    tol = (read t) :: Double
    lim = (read l) :: Int
    derivative = polyDerive poly
```

```
f = polyEval poly
fP = polyEval derivative
(value, txt) = newton f fP guess tol lim
msg = "Zero value of " ++ (show value) ++ " found.\n" ++ txt
```

4.14 Main Function

{{SWDDD MAIN MAIN}}

4.14.1 Requirements

- The main function **shall** {{SWDDD MAIN MAIN 10}} print a header message when first entered.
- The main function **shall** {{SWDDD MAIN MAIN 20}} print a prompt for input.
- The main function **shall** {{SWDDD MAIN MAIN 30}} accept a line of input.
- The line of input **shall** {{SWDDD MAIN MAIN 40}} be converted to uppercase.
- The main function **shall** {{SWDDD MAIN MAIN 50}} pass the input line to the command parser for interpretation.
- Polynomials **shall** {{SWDDD MAIN MAIN 60}} be stored in an array that can be indexed by name.
- The first word of the input line **shall** {{SWDDD MAIN MAIN 70}} be interpreted as the command.
- The ADD command **shall** {{SWDDD MAIN MAIN 80}} add two specified polynomials and produce a third.
- The DERIVE command **shall** {{SWDDD MAIN MAIN 90}} calculate the derivative of the specified polynomial.
- The DEL command **shall** {{SWDDD MAIN MAIN 100}} delete a polynomial from the saved list.
- The DIV command **shall** {{SWDDD MAIN MAIN 110}} divide two specified polynomials and produce a third.
- The EVAL command **shall** {{SWDDD MAIN MAIN 120}} compute the value of the specified polynomial at a specific value of X.
- The EXIT command **shall** {{SWDDD MAIN MAIN 130}} exit the program.
- The FACTOR command **shall** {{SWDDD MAIN MAIN 140}} produce the factors of a polynomial.
- The HELP command **shall** {{SWDDD MAIN MAIN 150}} produce a help message.
- The INTEGRATE command **shall** {{SWDDD MAIN MAIN 160}} produce the integral of the specified polynomial.

- The LIST command **shall** {{SWDDD MAIN MAIN 170}} produce a list of all the saved polynomials.
- The MUL command **shall** {{SWDDD MAIN MAIN 180}} multiply two specified polynomials and produce a third.
- The NEW command **shall** {{SWDDD MAIN MAIN 190}} create a new polynomial.
- The ROOT command **shall** {{SWDDD MAIN MAIN 200}} use various algorithms to find roots of the specified polynomial.
- The TABLE command **shall** {{SWDDD MAIN MAIN 210}} produce a table of values for the specified polynomial.
- The QUIT command **shall** {{SWDDD MAIN MAIN 220}} exit the program.

4.14.2 Function Definition

```
main = do
  putStrLn "Simple symbolic math package."
  mainLoop [("P1", [-1,2,3]), ("P2", [3,2,1])]
```

List of Trace Tags