

School of Computer Science and Software Engineering

CSE3322 Programming Languages and Implementation

Assignment 2

The structure should provide the following: Type definitions for `factor`, `term`, and `polynomial` which respectively represent a factor, term, and polynomial.

```
type factor = { var: string, power: int};
type term = { coef: real, factors: factor list};
type polynomial = term list;
```

Functions for constructing a factor, a term and a polynomial:

```
factor: string * int -> factor
term: real * factor list -> term
poly: term list -> polynomial
```

```
fun factor(s,p) = { var=s,power=p}:factor;
fun term(c,f) = {coef=c,factors=f}:term;
fun poly p = p:polynomial;
```

A function

`toString: polynomial -> string`

for returning a string describing a polynomial. **[1 mark]**

```
fun factorToString {var, power} =  
    " " ^ var ^ " " ^ (Int.toString power);  
  
fun factorsToString factors =  
    foldr (op ^) "" (map factorToString factors);  
  
fun termToString {coef, factors} =  
    (Real.toString coef) ^ (factorsToString factors);  
  
fun toString [] = "\n"  
  | toString [t] = termToString t  
  | toString (t::ts) =  
    (termToString t) ^ " + " ^ (toString ts);
```

A function

```
add: polynomial*polynomial -> polynomial
```

where `add(p1,p2)` is the polynomial obtained by adding the terms in `p1` and `p2`. It does not need to perform simplification. **[1 mark]**

```
fun add (p1,p2) = p1@p2;
```

A function

```
scale: real -> polynomial -> polynomial
```

where `scale r p` is the polynomial obtained by multiplying the coefficient of all terms in `p` by `r`. **[1 mark]**

```
fun scaleterm r {coef,factors}:term =  
    {coef=r*coef,factors=factors};  
fun scale r p = map (scaleterm r) p;
```

A function

```
eval: (string -> real) -> polynomial -> real
```

where `eval v p` is the result of evaluating `p` with `v`. [2 marks]

```
fun evalfactor v {var,power} =
    Real.Math.pow ((v var),real power);
fun evalterm v {coef,factors} =
    let val pf = map (evalfactor v) factors in
        (coef*(foldr (op * ) 1.0 pf))
    end;
fun eval v p =
    let val pe = map (evalterm v) p in
        (foldr (op +) 0.0 pe)
    end;
```

A function which takes a polynomial and returns a simplified version of the polynomial:

`simplify: polynomial -> polynomial`

The function should

- simplify terms by merging multiple variable occurrences in the same term into a single occurrence,
- simplify terms by removing factors with a power of 0,
- merge terms with the same factors into a single term and
- remove terms with 0.0 coefficients.

[2 marks]

```

(* some useful functions *)
(* filter those elements that do not
   satisfy a Boolean function from a list *)
fun filter f [] = []
  | filter f (x::xs) =
    if f(x) then x::(filter f xs)
    else filter f xs;

(* merge sort elements in a list given a
   comparison function *)
fun merge lt ([],M) = M
  | merge lt (L,[]) = L
  | merge lt (L as x::xs, M as y::ys) =
    if lt(x,y) then x::merge lt (xs,M)
    else y::merge lt (L,ys);

fun split [] = ([],[])
  | split [a] = ([a],[])
  | split (a::b::cs) =
    let
      val (M,N) = split(cs)
    in
      (a::M, b::N)
    end;

```

```
fun mergeSort lt [] = []
  | mergeSort lt [a] = [a]
  | mergeSort lt L =
      let
        val (M,N) = split L;
        val M = mergeSort lt M;
        val N = mergeSort lt N;
      in
        merge lt (M,N)
      end;
```

```

(* simplify a term by
  -- sorting the list of factors by variable name
  -- collapsing factors for the same variable
  -- removing factors with a zero power *)
fun factorlt (f1:factor,f2:factor) =
    #var f1 < #var f2;

fun collapseFactors [] = []
  | collapseFactors [f:factor] = [f]
  | collapseFactors (f1::f2::fs) =
    if #var f1 = #var f2 then
        collapseFactors (
            {var=(#var f1),
             power=(#power f1) + (#power f2)}::fs)
    else
        f1::(collapseFactors (f2::fs));

fun notZeroFactor(f:factor) = (#power f) <> 0;
fun rmZeroFactors fs = filter notZeroFactor fs;

fun simplifyTerm {coef, factors} =
    let val sf = (mergeSort factorlt factors) in
        {coef=coef,
         factors= rmZeroFactors (collapseFactors sf)}
    end;

```

```

(* simplify a polynomial by
-- simplifying the terms in the polynomial
-- sorting the list of terms by their factors
-- collapsing terms with the same factors
-- removing terms with a zero coefficient *)
fun factorslt (fs1,[]) = false
  | factorslt ([],fs2) = true
  | factorslt (f1::f1s,f2::f2s) =
      factorlt (f1,f2) orelse
      (not (factorlt (f2,f1)) andalso
      factorslt (f1s,f2s));

fun termlt (t1:term,t2:term) =
  factorslt (#factors t1, #factors t2);

fun collapseTerms [] = []
  | collapseTerms [t:term] = [t]
  | collapseTerms (t1::t2::ts) =
      if #factors t1 = #factors t2 then
        collapseTerms (
          {factors=(#factors t1),
           coef=(#coef t1) + (#coef t2)}::ts)
      else
        t1::(collapseTerms (t2::ts));

fun notZeroCoef(t:term) =
  (#coef t < ~1e~10) orelse (#coef t > 1e~10);
fun rmZeroCoefTerms ts = filter notZeroCoef ts;

```

```
fun simplify p =  
  let val sp = (map simplifyTerm p) in  
    rmZeroCoefTerms (  
      collapseTerms (mergeSort termlt sp))  
  end;
```

Define a **Polynomial** structure which groups the above functions and type definitions.

It should hide the definitions of the types and export only the functions detailed above (although you will need to define some hidden functions.) [1 mark]

```
signature POLYNOMIAL = sig
  type factor;
  type term;
  type polynomial;

  val factor: string * int -> factor;
  val term: real * factor list -> term;
  val poly: term list -> polynomial;
  val toString: polynomial -> string;
  val add: polynomial*polynomial -> polynomial;
  val scale: real -> polynomial -> polynomial;
  val eval: (string -> real) -> polynomial -> real;
  val simplify: polynomial -> polynomial
end;

structure Polynomial: POLYNOMIAL = struct

  ...
end;
```