AP CSP Create Task

```
#include <string>
     #include <stdio.h>
2
    #include <string.h>
3
    #include <vector>
    #include <iostream>
5
    #include <algorithm>
    #include <fstream>
    #include <stdlib.h>
    #include <math.h>
10
11
    using namespace std;
12
    ifstream fin("input.in");
13
    ofstream fout("output.out");
14
    struct expression
16
17
         bool type; // true for product, false for sum
18
19
20
21
         double coefficient;
22
23
         vector <struct expression> inside;
24
25
         // returns if expression evaluates to 0
26
         bool is_zero() const
27
28
             if(coefficient == 0)
29
30
                 return true;
31
32
            return false:
33
34
35
         // returns if function is a constant
36
37
         bool is_constant() const
38
             if(coefficient != 0 and power == 0)
39
40
41
                 return true;
42
            return false;
43
44
45
         // used by the sort functions, it just needs to put identical things next to each other
46
         bool operator< (const struct expression other) const
47
             if(inside.size() == other.inside.size()) // on this case, we want to separate expressions that are not equal from each other, but is_equal isn't defined...
49
50
51
                 return power < other.power;</pre>
52
53
             return inside.size() < other.inside.size();</pre>
54
55
    };
56
57
    // derivative functions
58
    struct expression product_rule(struct expression in);
     struct expression power_rule(struct expression in);
60
    struct expression chain_rule(struct expression in);
61
    struct expression derivative(struct expression in);
    struct\ expression\ derivative\_inside(struct\ expression\ in);
63
64
    // input functions
65
    int read_exp(void);
    struct expression read_in_expression(void);
67
68
69
    // comparison functions
    bool \ is\_equal(struct \ expression \ a, \ struct \ expression \ b);
70
    bool by_power(struct expression a, struct expression b);
71
72
    // output functions
     void output_expression(struct expression in);
74
    struct expression clean_up(struct expression in);
75
76
    // because-I-needed-it function
    int exp(int base, int power);
```

```
80
81
         // = <-- because some buttons on my keyboard don't work (for copying)</pre>
82
83
 84
         fin >> n; // takes in a number of expressions
 85
86
 87
         for(int i = 0; i < n; i++)</pre>
 88
             struct expression test:
89
 90
 91
             test = read_in_expression();
92
93
             test = clean_up(test); // just to make sure final answer is in a simpler form, which starts by making sure the input is in simplest form
 94
 95
             struct expression ans = derivative(test);
96
97
             output_expression(ans);
             fout << endl;
99
     }
100
101
102
     struct expression product_rule(struct expression in)
103
104
         struct expression ans;
105
106
         if(in.type != true)
107
108
              cout << "product rule was given a sum" << endl;</pre>
109
             exit(2);
110
111
112
         ans.type = false; // sum of the products, it's a holder for the result
113
         ans.coefficient = ans.power = 1; // just defaults
114
115
         struct expression terms; \ensuremath{//} this is the stuff multiplied together
116
         terms.inside = in.inside;
         terms.coefficient = terms.power = 1;
117
         terms.type = true;
118
119
120
         for(int i = 0; i < terms.inside.size(); i++) // applying the generalized product rule</pre>
121
             struct expression temp:
122
123
             temp = terms.inside[i];
124
             terms.inside[i] = derivative(temp);
125
126
127
              ans.inside.push_back(terms);
128
             terms.inside[i] = temp:
129
130
132
         return ans;
133
134
135
     struct expression power_rule(struct expression in)
136
137
         struct expression ans;
138
         if(in.power == 0) // it is a constant so just set everything to 0
139
140
             ans.power = 0:
141
             ans.coefficient = 0;
142
             return ans;
143
144
         ans.coefficient = in.coefficient * in.power; // apply the power rule
         ans.power = in.power - 1;
146
         ans.type = in.type;
147
148
         ans.inside = in.inside;
149
150
         return ans:
151
     }
152
153
     struct expression chain_rule(struct expression in)
154
155
         struct expression ans:
         ans.type = true; // product of f'(g(x)) and g'(x), so we need a new expression to contain them
156
157
158
         ans.inside.push_back(derivative_inside(in));
159
160
         in = power_rule(in); // only affects outer layer, f'(g(x))
161
162
         ans.inside.push_back(in);
```

```
163
164
          ans.coefficient = 1;
165
          ans.power = 1;
166
167
168
     }
169
170
     {\tt struct\ expression\ derivative}({\tt struct\ expression\ in})
171
172
          if(in.inside.size() == 0)
173
174
              return power_rule(in);
175
         else
176
177
178
              return chain_rule(in);
179
     }
180
181
182
     struct expression derivative_inside(struct expression in)
183
          struct expression ans;
184
185
          if(in.inside.size() == 0)
186
              ans.power = 0;
187
188
              ans.coefficient = 0;
189
              ans.type = false;
190
              return ans;
191
192
          if(in.type == false) // added together
193
              ans.type = false; // sum of the individual derivatives
194
195
              ans.coefficient = 1;
196
              ans.power = 1;
197
              for(int i = 0; i < in.inside.size(); i++)</pre>
198
                  ans.inside.push_back(derivative(in.inside[i]));
199
200
201
              return ans;
202
203
          if(in.type == true) // multiplied together
205
              // product rule...
              return product rule(in);
206
207
208
     }
209
210
     \verb"int read_exp(void) // \verb"just a simple helper function"
211
212
          int exponent;
213
          fin >> exponent:
214
          return exponent;
215
216
217
     \verb|struct expression read_in_expression(void)|\\
218
219
          char in;
          fin >> in;
220
221
222
223
224
          if((in >= '0' and in <= '9') or in == '-') // it's a number
225
226
              fin.putback(in);
              fin >> coefficient;
227
228
229
          else
230
          {
              coefficient = 1;
231
232
              fin.putback(in);
233
234
235
          char x;
236
237
238
          int exponent;
239
          struct expression ans;
240
          if(x != 'x' and x != '(')
241
242
243
              fin.putback(x);
244
              exponent = 0; // it's a constant
245
246
              ans.power = exponent;
```

```
247
              ans.coefficient = coefficient;
248
249
             return ans;
250
          }
251
          if(x == 'x')
252
253
              \ensuremath{//} return the expression with an exponent, do nothing, the code after will do that
254
255
256
          if(x == '(')
257
258
              // recursively read more expressions
259
             struct expression temp;
260
261
              temp = read_in_expression();
262
              ans.inside.push_back(temp);
263
264
265
              char next;
266
267
              do
268
269
                  fin >> next;
270
                  if(next != '+' and next != '-' and next !=')' and next != '*')
271
272
273
                      cout << "Expected *, +, - or )" << endl;
274
                      exit(-1);
275
276
277
                  if(next == '+')
278
                       if(ans.type == true \ and \ ans.inside.size() > 1) // conflicts with previous operation
279
280
                          cout << "Expected * but got -" << endl;</pre>
281
282
283
                       temp = read_in_expression();
284
                      ans.type = false;
285
286
287
                       ans.inside.push_back(temp);
288
289
                  if(next == '-')
290
291
                       if(ans.type == true \ and \ ans.inside.size() > 1) // conflicts with previous operation
292
                       {
                          cout << "Expected * but got -" << endl;</pre>
293
294
295
                       temp = read_in_expression();
                      temp.coefficient *= -1;
296
297
298
                       ans.type = false;
299
300
                      ans.inside.push_back(temp);
301
302
                  if(next == '*')
303
                       if(ans.type == false and ans.inside.size() > 1) // conflicts with previous operation
304
305
306
                          cout << "Expected +/- but got *" << endl;</pre>
307
308
309
                      temp = read_in_expression();
310
311
                      ans.type = true;
312
                       ans.inside.push_back(temp);
314
                  if(next == ')')
315
316
317
                      break;
318
              }while(next != ')');
319
320
321
322
          char carat;
323
          fin >> carat;
324
325
          if(carat != '^')
326
327
              fin.putback(carat);
328
329
              exponent = 1:
330
```

```
331
          else
332
          {
333
              exponent = read_exp();
334
335
          ans.power = exponent;
336
          ans.coefficient = coefficient;
337
338
          return ans;
339
340
341
342
     bool\ is\_equal(struct\ expression\ a,\ struct\ expression\ b)\ //\ only\ checks\ if\ stuff\ inside\ is\ equal
343
          if(a.inside.size() == 0 and b.inside.size() == 0)
344
345
346
347
          if(a.inside.size() != b.inside.size())
348
349
350
             return false;
351
352
353
          sort(a.inside.begin(),\ a.inside.end(),\ by\_power);\ //\ put\ stuff\ in\ order,\ puts\ equal\ stuff\ together
354
          sort(b.inside.begin(), b.inside.end(), by_power);
355
356
          for(int i = 0; i < a.inside.size(); i++)</pre>
357
              if(a.inside[i].coefficient != b.inside[i].coefficient)
358
359
360
                  return false;
361
              if(a.inside[i].power != b.inside[i].power)
362
363
364
                  return false;
365
              if(is_equal(a.inside[i], b.inside[i]) == false)
366
367
368
                  return false;
369
             }
          }
370
371
          return true;
373
374
375
     bool by_power(struct expression a, struct expression b)
376
          if(a.power == b.power)
377
378
379
             return a.coefficient < b.coefficient;</pre>
380
381
          return a.power > b.power:
382
383
384
      void output expression(struct expression in)
385
386
          in = clean_up(in);
387
          sort(in.inside.begin(), in.inside.end(), by_power);
          if(in.coefficient == 0) // should have been cleaned up already? but just in case
388
389
390
391
              return;
392
393
          if(in.power == 0)
394
              fout << in.coefficient;</pre>
395
396
              return;
397
398
          if(in.inside.size() == 0)
399
400
401
              if(in.power == 0)
402
                  fout << in.coefficient;</pre>
403
404
405
              else
406
407
                  if(in.coefficient != 1)
409
                      fout << in.coefficient;</pre>
410
411
                  fout << "x";
412
                  if(in.power != 1)
413
                      fout << "^" << in.power;
414
```

```
415
416
417
             return;
418
419
         if(in.coefficient != 1)
420
421
422
              fout << in.coefficient;</pre>
423
424
         if(in.power == 0) // only care about coefficient
425
426
427
             return;
428
429
430
         if(in.inside.size() > 1)
431
             fout << "(";
432
433
         for(int i = 0; i < in.inside.size(); i++) // recursively print stuff inside</pre>
434
435
              output_expression(in.inside[i]);
              if(i < in.inside.size() - 1)</pre>
436
437
438
                  if(in.type)
439
                  {
                      fout << " * ";
440
441
442
                  else
443
                  {
444
                      fout << " + ";
445
446
             }
447
448
         if(in.inside.size() > 1)
449
             fout << ")";
450
451
         if(in.power != 1)
452
              fout << "^" << in.power;
453
454
455
456
457
     struct expression clean up(struct expression in)
458
459
         for(int i = 0; i < in.inside.size(); i++)</pre>
460
             clean_up(in.inside[i]); // clean up those first
461
462
463
         if(in.inside.size() == 1) // bring everything up a level
464
465
466
              in.coefficient *= exp(in.inside[0].coefficient, in.power);
467
              in.power *= in.inside[0].power;
468
              in.inside = in.inside[0].inside;
469
470
471
             return in;
472
473
474
         if(in.type == true) // product
475
476
              // clean constants
477
              for(int i = 0; i < in.inside.size(); i++)</pre>
478
479
                  if(in.inside[i].is\_constant())
480
481
                      in.coefficient *= exp(in.inside[0].coefficient, in.power);
482
                      in.inside.erase(in.inside.begin() + i);
483
484
485
486
487
488
              // could also merge stuff that is the same base
489
              sort(in.inside.begin(), in.inside.end());
490
491
492
              for(int i = 1; i < in.inside.size(); i++)</pre>
493
                  if(is\_equal(in.inside[i],\ in.inside[i-1]))\ //\ they\ have\ the\ same\ base
494
495
496
                      in.inside[i].power += in.inside[i - 1].power;
                      in.inside[i].coefficient *= in.inside[i - 1].coefficient;
497
498
```

```
499
                                                                     in.inside.erase(in.inside.begin() + i - 1);
 500
 501
                                                       }
 502
                                         }
 503
 504
                                          return in;
 505
                              }
 506
 507
                               else // sum
 508
                               {
                                           // clean zeroes
509
 510
                                           for(int i = 0; i < in.inside.size(); i++)</pre>
 511
512
                                                       if(in.inside[i].is_zero())
 513
 514
                                                                     in.inside.erase(in.inside.begin() + i);
 515
                                                                    i--;
                                                      }
 516
 517
 518
                                          // could also merge stuff that is the same base and power
 519
 520
 521
                                           sort(in.inside.begin(), in.inside.end());
 522
                                           for(int i = 1; i < in.inside.size(); i++)</pre>
523
 524
 525
                                                        \textbf{if(is\_equal(in.inside[i], in.inside[i-1]) and in.inside[i].power == in.inside[i-1].power) // they have the same base and power interval of the same base and power interv
 526
527
 528
                                                                     in.inside[i].coefficient += in.inside[i - 1].coefficient;
 529
                                                                    in.inside.erase(in.inside.begin() + i - 1);
530
531
 532
 533
                                         }
534
 535
                                         return in;
 536
                 }
 537
538
 539
                 int exp(int base, int power) // naive exponent function
 541
                               int ans = 1:
542
 543
                               for(int i = 0; i < power; i++)</pre>
 544
                                          ans *= base;
545
 546
 547
                               return ans;
                }
 548
 549
 550
 551
```

PDF document made with CodePrint using Prism