

AP CSP Create Task

```

1  #include <string>
2  #include <stdio.h>
3  #include <string.h>
4  #include <vector>
5  #include <iostream>
6  #include <algorithm>
7  #include <fstream>
8  #include <stdlib.h>
9  #include <math.h>
10 #include <iomanip>
11 using namespace std;
12
13 ifstream fin("input.in");
14 ofstream fout("output.out");
15
16 struct expression
17 {
18     bool type; // true for product, false for sum
19
20     double power;
21
22     double coefficient;
23
24     vector <struct expression> inside;
25
26     // returns if expression evaluates to 0
27     bool is_zero() const
28     {
29         if(coefficient == 0)
30         {
31             return true;
32         }
33         return false;
34     }
35
36     // returns if function is a constant
37     bool is_constant() const
38     {
39         if(coefficient != 0 and power == 0)
40         {
41             return true;
42         }
43         return false;
44     }
45
46     // used by the sort functions, it just needs to put identical things next to each other
47     bool operator< (const struct expression other) const
48     {
49         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...
50         {
51             return power < other.power;
52         }
53
54         return inside.size() < other.inside.size();
55     }
56 };
57
58 // derivative functions
59 struct expression product_rule(struct expression in);
60 struct expression power_rule(struct expression in);
61 struct expression chain_rule(struct expression in);
62 struct expression derivative(struct expression in);
63 struct expression derivative_inside(struct expression in);
64
65 // input functions
66 int read_exp(void);
67 struct expression read_in_expression(void);
68
69 // comparison functions
70 bool is_equal(struct expression a, struct expression b);
71 bool by_power(struct expression a, struct expression b);
72
73 // output functions
74 void output_expression(struct expression in);
75 struct expression clean_up(struct expression in);
76
77 // because-I-needed-it function
78 int exp(int base, int power);

```

```

79
80 int main(void)
81 {
82     // = <-- because some buttons on my keyboard don't work (for copying)
83
84     int n;
85     fin >> n; // takes in a number of expressions
86
87     for(int i = 0; i < n; i++)
88     {
89         struct expression test;
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);
98         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;
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; // this is the stuff multiplied together
116     terms.inside = in.inside;
117     terms.coefficient = terms.power = 1;
118     terms.type = true;
119
120     for(int i = 0; i < terms.inside.size(); i++) // applying the generalized product rule
121     {
122         struct expression temp;
123         temp = terms.inside[i];
124
125         terms.inside[i] = derivative(temp);
126
127         ans.inside.push_back(terms);
128
129         terms.inside[i] = temp;
130     }
131
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
145     ans.coefficient = in.coefficient * in.power; // apply the power rule
146     ans.power = in.power - 1;
147     ans.type = in.type;
148     ans.inside = in.inside;
149
150     return ans;
151 }
152
153 struct expression chain_rule(struct expression in)
154 {
155     struct expression ans;
156     ans.type = true; // product of f'(g(x)) and g'(x), so we need a new expression to contain them
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     return ans;
168 }
169
170 struct expression derivative(struct expression in)
171 {
172     if(in.inside.size() == 0)
173     {
174         return power_rule(in);
175     }
176     else
177     {
178         return chain_rule(in);
179     }
180 }
181
182 struct expression derivative_inside(struct expression in)
183 {
184     struct expression ans;
185     if(in.inside.size() == 0)
186     {
187         ans.power = 0;
188         ans.coefficient = 0;
189         ans.type = false;
190         return ans;
191     }
192     if(in.type == false) // added together
193     {
194         ans.type = false; // sum of the individual derivatives
195         ans.coefficient = 1;
196         ans.power = 1;
197         for(int i = 0; i < in.inside.size(); i++)
198         {
199             ans.inside.push_back(derivative(in.inside[i]));
200         }
201         return ans;
202     }
203     if(in.type == true) // multiplied together
204     {
205         // product rule...
206         return product_rule(in);
207     }
208 }
209
210 int read_exp(void) // just a simple helper function
211 {
212     int exponent;
213     fin >> exponent;
214     return exponent;
215 }
216
217 struct expression read_in_expression(void)
218 {
219     char in;
220     fin >> in;
221
222     int coefficient;
223
224     if((in >= '0' and in <= '9') or in == '-') // it's a number
225     {
226         fin.putback(in);
227         fin >> coefficient;
228     }
229     else
230     {
231         coefficient = 1;
232         fin.putback(in);
233     }
234
235     char x;
236     fin >> x;
237
238     int exponent;
239     struct expression ans;
240
241     if(x != 'x' and x != '(')
242     {
243         fin.putback(x);
244
245         exponent = 0; // it's a constant
246         ans.power = exponent;

```

```
247     ans.coefficient = coefficient;
248
249     return ans;
250 }
251
252 if(x == 'x')
253 {
254     // return the expression with an exponent, do nothing, the code after will do that
255 }
256 if(x == '(')
257 {
258     // recursively read more expressions
259
260     struct expression temp;
261     temp = read_in_expression();
262
263     ans.inside.push_back(temp);
264
265     char next;
266
267     do
268     {
269         fin >> next;
270
271         if(next != '+' and next != '-' and next != ')' and next != '*')
272         {
273             cout << "Expected *, +, - or )" << endl;
274             exit(-1);
275         }
276
277         if(next == '+')
278         {
279             if(ans.type == true and ans.inside.size() > 1) // conflicts with previous operation
280             {
281                 cout << "Expected * but got +" << endl;
282             }
283             temp = read_in_expression();
284
285             ans.type = false;
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             {
293                 cout << "Expected * but got -" << endl;
294             }
295             temp = read_in_expression();
296             temp.coefficient *= -1;
297
298             ans.type = false;
299
300             ans.inside.push_back(temp);
301         }
302         if(next == '*')
303         {
304             if(ans.type == false and ans.inside.size() > 1) // conflicts with previous operation
305             {
306                 cout << "Expected +/- but got *" << endl;
307             }
308
309             temp = read_in_expression();
310
311             ans.type = true;
312
313             ans.inside.push_back(temp);
314         }
315         if(next == ')')
316         {
317             break;
318         }
319     }while(next != ')');
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 {
344     if(a.inside.size() == 0 and b.inside.size() == 0)
345     {
346         return true;
347     }
348     if(a.inside.size() != b.inside.size())
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++)
357     {
358         if(a.inside[i].coefficient != b.inside[i].coefficient)
359         {
360             return false;
361         }
362         if(a.inside[i].power != b.inside[i].power)
363         {
364             return false;
365         }
366         if(is_equal(a.inside[i], b.inside[i]) == false)
367         {
368             return false;
369         }
370     }
371
372     return true;
373 }
374
375 bool by_power(struct expression a, struct expression b)
376 {
377     if(a.power == b.power)
378     {
379         return a.coefficient < b.coefficient;
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);
388     if(in.coefficient == 0) // should have been cleaned up already? but just in case
389     {
390         fout << "0";
391         return;
392     }
393     if(in.power == 0)
394     {
395         fout << in.coefficient;
396         return;
397     }
398
399     if(in.inside.size() == 0)
400     {
401         if(in.power == 0)
402         {
403             fout << in.coefficient;
404         }
405         else
406         {
407             if(in.coefficient != 1)
408             {
409                 fout << in.coefficient;
410             }
411             fout << "x";
412             if(in.power != 1)
413             {
414                 fout << "^" << in.power;

```

```

415     }
416 }
417 return;
418 }
419
420 if(in.coefficient != 1)
421 {
422     fout << in.coefficient;
423 }
424
425 if(in.power == 0) // only care about coefficient
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
434 {
435     output_expression(in.inside[i]);
436     if(i < in.inside.size() - 1)
437     {
438         if(in.type)
439         {
440             fout << " * ";
441         }
442         else
443         {
444             fout << " + ";
445         }
446     }
447 }
448 if(in.inside.size() > 1)
449     fout << ")";
450
451 if(in.power != 1)
452 {
453     fout << "^" << in.power;
454 }
455 }
456
457 struct expression clean_up(struct expression in)
458 {
459     for(int i = 0; i < in.inside.size(); i++)
460     {
461         clean_up(in.inside[i]); // clean up those first
462     }
463
464     if(in.inside.size() == 1) // bring everything up a level
465     {
466         in.coefficient *= exp(in.inside[0].coefficient, in.power);
467         in.power *= in.inside[0].power;
468
469         in.inside = in.inside[0].inside;
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++)
478         {
479             if(in.inside[i].is_constant())
480             {
481                 in.coefficient *= exp(in.inside[i].coefficient, in.power);
482
483                 in.inside.erase(in.inside.begin() + i);
484                 i--;
485             }
486         }
487
488         // could also merge stuff that is the same base
489
490         sort(in.inside.begin(), in.inside.end());
491
492         for(int i = 1; i < in.inside.size(); i++)
493         {
494             if(is_equal(in.inside[i], in.inside[i - 1])) // they have the same base
495             {
496                 in.inside[i].power += in.inside[i - 1].power;
497                 in.inside[i].coefficient *= in.inside[i - 1].coefficient;
498             }
499         }
500     }

```

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