

Experimentos VPL AED-I

[Dashboard](#) / [My courses](#) / [experimentos-vpl-aedi](#) / [Estruturas lineares: listas ligadas \(parte 2\), pilhas e filas](#) / [\[EP\] Pilha dinâmica](#)

Description

Submissions list

Similarity

Test activity

[EP] Pilha dinâmica

Due date: Sunday, 19 March 2023, 3:59 PM

Requested files: pilha.c ([Download](#))

Type of work: Individual work

Grade settings: Maximum grade: 1 Hidden

Visible: No

Run: No. **Run script:** C. **Evaluate:** Yes

Automatic grade: Yes.

Implemente as funções de uma **Pilha dinâmica** de números inteiros (implementada com lista ligada):

```
PilhaDinamica *criar_pilha();
int empilhar(PilhaDinamica *pilha, int valor);
int desempilhar(PilhaDinamica *pilha, int *valor);
void liberar_pilha(PilhaDinamica *pilha);
```

Considere a seguinte estrutura para uma Pilha:

```
typedef struct PilhaDinamica PilhaDinamica;
struct PilhaDinamica {
    LinkedNode *topo;
};
```

O retorno das funções empilhar e desempilhar indica se operação foi executada com sucesso. Por exemplo, desempilhar pode retornar 0 para pilha vazia. Se a operação foi executada com sucesso, retorna 1.

Considere a seguinte estrutura para um nó de lista ligada:

```
typedef struct LinkedNode LinkedNode;
struct LinkedNode {
    int data;
    LinkedNode *next;
};
```

Importante: submeta apenas as funções da pilha. Não use variáveis globais. Não inclua o main. Não use printf/fprintf/puts/gets/fgets/scanf/fscanf e headers adicionais (por exemplo, stdio.h). Neste exercício, pode usar stdlib.h.

A definição da struct LinkedNode e da struct PilhaDinamica já existe no sistema de correção automática (portanto, não inclua a definição dessas structs no código submetido). É necessário incluir a seguinte linha no início do código submetido:

```
#include "lista.h"
#include "pilha.h"
```

Formato do caso de teste: esse é o formato dos casos de teste que aparecem ao avaliar a atividade; não inclua impressão de dados no código, essa impressão é feita automaticamente pelo sistema de correção de acordo com o retorno da função submetida.

Entrada:

- operações sobre a pilha:

```
E 10 (empilhar o 10)
D (desempilhar)
S (sair)
```

Saída:

- resultado das operações

Requested files

pilha.c

```

1  #include <stdlib.h>
2  #include "lista.h"
3  #include "pilha.h"
4
5  PilhaDinamica *criar_pilha() {
6      //Codigo da funcao aqui
7  }
8
9  int empilhar(PilhaDinamica *pilha, int valor) {
10     //Codigo da funcao aqui
11 }
12
13 int desempilhar(PilhaDinamica *pilha, int *valor) {
14     //Codigo da funcao aqui
15 }
16
17 void liberar_pilha(PilhaDinamica *pilha) {
18     //Codigo da funcao aqui
19 }

```

Execution files

vpl_run.sh

```

1  if egrep -zq 'printf|puts|fprintf|gets|fgets|scanf|fscanf|stdio' $VPL_SUBFILE0 ; then
2      echo "Nao eh permitido utilizar printf/puts/fprintf/gets/fgets/scanf/fscanf e stdio.h." >> vpl_compilation_error.txt
3      exit 0
4  fi
5
6  eval gcc -fno-diagnostics-color -o vpl_execution $2 vpl_test_main.c $VPL_SUBFILE0
7
8  chmod +x vpl_execution

```

vpl_evaluate.sh

```

1  # [14/10/2020 - modified by Paulo Pisani] - Added code to display compilation error in comments section - this code is based on branch 3.3.8
2  # of default_evaluate.sh at https://github.com/jcrodriguez-dis/moodle-mod_vpl/tree/v3.3.8/jail/default_scripts (download 13/10/2020)
3  # from Juan Carlos Rodriguez-del-Pino, 2014)
4  # License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
5  # License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
6
7  # Set to 1 to show compilation output in comments section
8  export VPLX_SHOW_COMPILATION_ERROR_OUTPUT=1
9
10 #!/bin/bash
11 # This file is part of VPL for Moodle
12 # Default evaluate script for VPL
13 # Copyright (C) 2014 onwards Juan Carlos Rodriguez-del-Pino
14 # License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
15 # Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>
16
17 #load VPL environment vars
18 . common_script.sh
19 if [ "$SECONDS" = "" ] ; then
20     export SECONDS=20
21 fi
22 let VPL_MAXTIME=$SECONDS-5;
23 if [ "$VPL_GRADEMIN" = "" ] ; then
24     export VPL_GRADEMIN=0
25     export VPL_GRADEMAX=10
26 fi
27
28 #exist run script?
29 if [ ! -s vpl_run.sh ] ; then
30     echo "I'm sorry, but I haven't a default action to evaluate the type of submitted files"
31 else
32     #avoid conflict with C++ compilation
33     mv vpl_evaluate.cpp vpl_evaluate.cpp.save
34     #Prepare run
35     ./vpl_run.sh &>>vpl_compilation_error.txt
36     cat vpl_compilation_error.txt
37     if [ -f vpl_execution ] ; then
38         mv vpl_execution vpl_test
39         if [ -f vpl_evaluate.cases ] ; then
40             mv vpl_evaluate.cases evaluate.cases
41         else
42             echo "Error need file 'vpl_evaluate.cases' to make an evaluation"
43             exit 1
44         fi
45         mv vpl_evaluate.cpp.save vpl_evaluate.cpp
46         check_program g++
47         g++ vpl_evaluate.cpp -g -lm -lutil -o .vpl_tester
48         if [ ! -f .vpl_tester ] ; then
49             echo "Error compiling evaluation program"
50             exit 1
51         else
52             cat vpl_environment.sh >> vpl_execution
53             echo ".vpl_tester" >> vpl_execution
54         fi
55     else
56         echo "#!/bin/bash" >> vpl_execution
57         echo "echo" >> vpl_execution
58         echo "echo '<!--'" >> vpl_execution
59         echo "echo '-$VPL_COMPILATIONFAILED'" >> vpl_execution
60         if [ -f vpl_wexecution ] ; then
61             echo "echo '====='" >> vpl_execution
62             echo "echo 'It seems you are trying to test a program with a graphic user interface'" >> vpl_execution
63         fi
64
65         if [ $VPLX_SHOW_COMPILATION_ERROR_OUTPUT -eq 1 ] ; then
66             echo "cat vpl_compilation_error.txt" >> vpl_execution
67         fi
68
69         echo "echo '-->" >> vpl_execution
70         echo "echo" >> vpl_execution
71         echo "echo 'Grade :>=$VPL_GRADEMIN'" >> vpl_execution
72     fi
73     chmod +x vpl_execution
74 fi
75

```

vpl_evaluate.cases

```
1 case=Test 1
2 grade reduction=100%
3 input=E 10
4 E 20
5 D
6 E 30
7 D
8 D
9 D
10 E 40
11 E 50
12 E 60
13 D
14 S
15
16
17 output="E 10
18 E 20
19 D 20
20 E 30
21 D 30
22 D 10
23 Pilha vazia
24 E 40
25 E 50
26 E 60
27 D 60
28 "
29
30 case=Test 2
31 grade reduction=100%
32 input=D
33 E 11
34 E 22
35 E 66
36 E 88
37 D
38 D
39 E 40
40 E 50
41 D
42 D
43 D
44 D
45 D
46 S
47
48
49 output="Pilha vazia
50 E 11
51 E 22
52 E 66
53 E 88
54 D 88
55 D 66
56 E 40
57 E 50
58 D 50
59 D 40
60 D 22
61 D 11
62 Pilha vazia
63 "
64
65 case=Test 3
66 grade reduction=100%
67 input=D
68 D
69 S
70
71 output="Pilha vazia
72 Pilha vazia
73 "
74
75 case=Test 4
76 grade reduction=100%
77 input=E 10
78 E 20
79 E 30
80 E 40
81 E 90
82 E 100
83 S
84
85 output="E 10
86 E 20
87 E 30
88 E 40
89 E 90
90 E 100
91 "
92
93 case=Test 5
94 grade reduction=100%
95 input=E 10
96 E 20
97 E 30
98 E 40
99 E 90
100 E 100
101 D
102 D
103 D
104 E 500
105 E 501
106 E 502
107 E 503
108 E 504
109 E 506
110 E 507
111 E 508
112 E 509
113 E 510
114 E 511
115 E 512
116 E 513
117 E 514
118 E 515
119 E 516
120 E 517
121 E 518
122 E 519
123 E 520
```

```
124 E 521
125 D
126 D
127 D
128 D
129 D
130 D
131 D
132 E 700
133 E 250
134 D
135 D
136 D
137 S
138
139 output="E 10
140 E 20
141 E 30
142 E 40
143 E 90
144 E 100
145 D 100
146 D 90
147 D 40
148 E 500
149 E 501
150 E 502
151 E 503
152 E 504
153 E 506
154 E 507
155 E 508
156 E 509
157 E 510
158 E 511
159 E 512
160 E 513
161 E 514
162 E 515
163 E 516
164 E 517
165 E 518
166 E 519
167 E 520
168 E 521
169 D 521
170 D 520
171 D 519
172 D 518
173 D 517
174 D 516
175 D 515
176 E 700
177 E 250
178 D 250
179 D 700
180 D 514
181 "
```

vpl_evaluate.cpp

```

1  /**
2  * [12/11/2020 - modified by Paulo Pisani] - Changed call to valgrind: uses /usr/bin/valgrind instead of /bin/bash + valgrind
3  * [14/10/2020 - modified by Paulo Pisani] - Added code to check memory leak using valgrind - this code is based on branch 3.3.8 of vpl_evaluate.cpp
4  * at https://github.com/jcrodriguez-dis/moodle-mod_vpl/tree/v3.3.8/jail/default_scripts (download 13/10/2020)
5  * from Juan Carlos Rodriguez-del-Pino, 2019
6  * License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
7  */
8
9  const bool CHECK_LEAK_FLAG = true;
10 const float MEMORY_LEAK_REDUCTION = 0.2;
11
12 /**
13 * VPL builtin program for submissions evaluation
14 * Copyright (C) 2019 Juan Carlos Rodriguez-del-Pino
15 * License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
16 * Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>
17 */
18
19 #include <cstdlib>
20 #include <cstdio>
21 #include <climits>
22 #include <limits>
23 #include <errno.h>
24 #include <sys/types.h>
25 #include <sys/wait.h>
26 #include <poll.h>
27 #include <unistd.h>
28 #include <pty.h>
29 #include <fcntl.h>
30 #include <signal.h>
31 #include <cstring>
32 #include <string>
33 #include <iostream>
34 #include <sstream>
35 #include <vector>
36 #include <cmath>
37 #include <execinfo.h>
38 #include <regex.h>
39 #include <string>
40
41 using namespace std;
42
43 const int MAXCOMMENTS = 20;
44 const int MAXCOMMENTSLENGTH = 100*1024;
45 const int MAXCOMMENTSITLENGTH = 1024;
46 const int MAXOUTPUT = 256* 1024 ;//256Kb
47
48 /**
49 * Class Tools Declaration
50 */
51 class Tools {
52 public:
53     static bool existFile(string name);
54     static string readFile(string name);
55     static vector<string> splitLines(const string &data);
56     static int nextLine(const string &data);
57     static string caseFormat(string text);
58     static string toLower(const string &text);
59     static string normalizeTag(const string &text);
60     static bool parseLine(const string &text, string &name, string &data);
61     static string trimRight(const string &text);
62     static string trim(const string &text);
63     static void fdblock(int fd, bool set);
64     static bool convert2(const string& str, double &data);
65     static bool convert2(const string& str, long int &data);
66     static const char* getenv(const char* name, const char* defaultvalue);
67     static double getenv(const char* name, double defaultvalue);
68 };
69
70 /**
71 * Class Stop Declaration
72 */
73 class Stop{
74     static volatile bool TERMRequested;
75 public:
76     static void setTERMRequested();
77     static bool isTERMRequested();
78 };
79
80 /**
81 * Class Timer Declaration
82 */
83 class Timer{
84     static time_t startTime;
85 public:
86     static void start();
87     static int elapsedTime();
88 };
89
90 /**
91 * Class I18n Declaration
92 */
93 class I18n{
94 public:
95     void init();
96     const char *get_string(const char *s);
97 };
98
99 /**
100 * Interface OutputChecker
101 */
102 class OutputChecker{
103 protected:
104     string text;
105
106 public:
107     OutputChecker(const string &t):text(t){}
108     virtual ~OutputChecker(){};
109     virtual string type(){return "";}
110     virtual operator string (){return "";}
111     virtual string outputExpected(){return text;}
112     virtual string studentOutputExpected(){return text;}
113     virtual bool match(const string&)=0;
114     virtual OutputChecker* clone()=0;
115 };
116
117 /**
118 * Class NumbersOutput Declaration
119 */
120 class NumbersOutput:public OutputChecker{
121     struct Number{
122         bool isInteger;
123         long int integer;

```

```

124         double científico;
125
126         bool set(const string& str);
127         bool operator==(const Number &o)const;
128         bool operator!=(const Number &o)const;
129         operator string () const;
130     };
131
132     vector<Number> numbers;
133     bool startWithAsterisk;
134     string cleanText;
135
136     static bool isNum(char c);
137     static bool isNumStart(char c);
138     bool calcStartWithAsterisk();
139
140 public:
141     NumbersOutput(const string &text);//:OutputChecker(text);
142     string studentOutputExpected();
143     bool operator==(const NumbersOutput& o)const;
144     bool match(const string& output);
145     OutputChecker* clone();
146     static bool typeMatch(const string& text);
147     string type();
148     operator string () const;
149 };
150
151 /**
152  * Class TextOutput Declaration
153  */
154 class TextOutput:public OutputChecker{
155     vector<string> tokens;
156     bool isAlpha(char c);
157
158 public:
159     TextOutput(const string &text);//:OutputChecker(text);
160     bool operator==(const TextOutput& o);
161     bool match(const string& output);
162     OutputChecker* clone();
163     static bool typeMatch(const string& text);
164     string type();
165 };
166
167 /**
168  * Class ExactTextOutput Declaration
169  */
170 class ExactTextOutput:public OutputChecker{
171     string cleanText;
172     bool startWithAsterisk;
173     bool isAlpha(char c);
174
175 public:
176     ExactTextOutput(const string &text);//:OutputChecker(text);
177     string studentOutputExpected();
178     bool operator==(const ExactTextOutput& o);
179     bool match(const string& output);
180     OutputChecker* clone();
181     static bool typeMatch(const string& text);
182     string type();
183 };
184
185 /**
186  * Class RegularExpressionOutput Declaration
187  * Regular Expressions implemented by:
188  * Daniel José Ojeda Loisel
189  * Juan David Vega Rodriguez
190  * Miguel Ángel Viera González
191  */
192 class RegularExpressionOutput:public OutputChecker {
193     string errorCase;
194     string cleanText;
195     regex_t expression;
196     bool flagI;
197     bool flagM;
198     int reti;
199
200 public:
201     RegularExpressionOutput (const string &text, const string &actualCaseDescription);
202
203     bool match (const string& output);
204     // Regular Expression compilation (with flags in mind) and comparison with the input and output evaluation
205
206     string studentOutputExpected();
207     // Returns the expression without flags nor '/'
208
209     OutputChecker* clone();
210
211     static bool typeMatch(const string& text);
212     // Tests if it's a regular expression. A regular expressions should be between ../
213
214     string type();
215 };
216 /**
217  * Class Case Declaration
218  * Case represents cases
219  */
220 class Case {
221     string input;
222     vector< string > output;
223     string caseDescription;
224     float gradeReduction;
225     string failMessage;
226     string programToRun;
227     string programArgs;
228     int expectedExitCode; // Default value std::numeric_limits<int>::min()
229     string variation;
230     bool checkLeak;
231
232 public:
233     Case();
234     void reset();
235     void addInput(string );
236     string getInput();
237     void addOutput(string );
238     const vector< string > & getOutput();
239     void setFailMessage(const string &);
240     string getFailMessage();
241     void setCaseDescription(const string &);
242     string getCaseDescription();
243     void setGradeReduction(float);
244     float getGradeReduction();
245     void setExpectedExitCode(int);
246     int getExpectedExitCode();
247     void setProgramToRun(const string &);

```

```

247     string getProgramToRun();
248     void setProgramArgs(const string &);
249     string getProgramArgs();
250     void setVariation(const string &);
251     string getVariation();
252     void setCheckLeak(bool cl);
253     bool getCheckLeak();
254 };
255
256 /**
257  * Class TestCase Declaration
258  * TestCase represents cases to tested
259  */
260 class TestCase {
261     const char *command;
262     const char **argv;
263     static const char **envv;
264     int id;
265     bool correctOutput;
266     bool checkLeak;
267     bool hasLeak;
268     bool outputTooLarge;
269     bool programTimeout;
270     bool executionError;
271     bool correctExitCode;
272     char executionErrorReason[1000];
273     int sizeReaded;
274     string input;
275     vector< OutputChecker* > output;
276     string caseDescription;
277     float gradeReduction;
278     float gradeReductionApplied;
279     string failMessage;
280     string programToRun;
281     string programArgs;
282     string variation;
283     int expectedExitCode; // Default value std::numeric_limits<int>::min()
284     int exitCode; // Default value std::numeric_limits<int>::min()
285     string programOutputBefore, programOutputAfter, programInput;
286
287     void cutOutputTooLarge(string &output);
288     void readWrite(int fdread, int fdwrite);
289     void addOutput(const string &o, const string &actualCaseDescription);
290     void runTestExec(time_t timeout, bool check_leak_only);
291     void removeZerosFromVector(char buf[], int len);
292 public:
293     static void setEnvironment(const char **environment);
294     void setDefaultCommand();
295     TestCase(const TestCase &o);
296     TestCase& operator=(const TestCase &o);
297     ~TestCase();
298     TestCase(int id, const string &input, const vector<string> &output,
299             const string &caseDescription, const float gradeReduction,
300             string failMessage, string programToRun, string programArgs, int expectedExitCode, bool cl);
301     bool isCorrectResult();
302     bool hasMemoryLeak();
303     bool isExitCodeTested();
304     float getGradeReduction();
305     void setGradeReductionApplied(float r);
306     float getGradeReductionApplied();
307     string getCaseDescription();
308     string getCommentTitle(bool withGradeReduction/*=false*/); // Suui
309     string getComment();
310     void splitArgs(string programArgs, bool checkLeakOnly);
311     void runTest(time_t timeout);
312     bool match(string data);
313     bool matchMemoryLeak(string data);
314 };
315
316 /**
317  * Class Evaluation Declaration
318  */
319 class Evaluation {
320     int maxtime;
321     float grademin, grademax;
322     string variation;
323     bool noGrade;
324     float grade;
325     int nerrors, nruns;
326     vector<TestCase> testCases;
327     char comments[MAXCOMMENTS + 1][MAXCOMMENTSLENGTH + 1];
328     char titles[MAXCOMMENTS + 1][MAXCOMMENTSSTITLELENGTH + 1];
329     char titlesGR[MAXCOMMENTS + 1][MAXCOMMENTSSTITLELENGTH + 1];
330     volatile int ncomments;
331     volatile bool stopping;
332     static Evaluation *singlenton;
333     Evaluation();
334
335 public:
336     static Evaluation* getSinglenton();
337     static void deleteSinglenton();
338     void addTestCase(Case &);
339     void removeLastNL(string &s);
340     bool cutToEndTag(string &value, const string &endTag);
341     void loadTestCases(string fname);
342     void loadParams();
343     void addFatalError(const char *m);
344     void runTests();
345     void outputEvaluation();
346 };
347
348 ////////////////////////////////////////
349 ////////////////////////////////////////
350 //////////////////////////////////////// END OF DECLARATIONS ////////////////////////////////////////
351 ////////////////////////////////////////
352 ////////////////////////////////////////
353
354 ////////////////////////////////////////
355 ////////////////////////////////////////
356 //////////////////////////////////////// BEGINNING OF DEFINITIONS ////////////////////////////////////////
357 ////////////////////////////////////////
358 ////////////////////////////////////////
359
360 volatile bool Stop::TERMRequested = false;
361 time_t Timer::startTime;
362 const char **TestCase::envv=NULL;
363 Evaluation* Evaluation::singlenton = NULL;
364
365 /**
366  * Class Tools Definitions
367  */
368
369 bool Tools::existFile(string name) {

```

```

370 FILE *f = fopen(name.c_str(), "r");
371 if (f != NULL) {
372     fclose(f);
373     return true;
374 }
375 return false;
376 }
377
378 string Tools::readFile(string name) {
379     char buf[1000];
380     string res;
381     FILE *f = fopen(name.c_str(), "r");
382     if (f != NULL)
383         while (fgets(buf, 1000, f) != NULL)
384             res += buf;
385     return res;
386 }
387
388 vector<string> Tools::splitLines(const string &data) {
389     vector<string> lines;
390     int len, l = data.size();
391     int startLine = 0;
392     char pc = 0, c;
393     for (int i = 0; i < l; i++) {
394         c = data[i];
395         if (c == '\n') {
396             len = i - startLine;
397             if (pc == '\r')
398                 len--;
399             lines.push_back(data.substr(startLine, len));
400             startLine = i + 1;
401         }
402         pc = c;
403     }
404     if (startLine < l) {
405         len = l - startLine;
406         if (pc == '\r')
407             len--;
408         lines.push_back(data.substr(startLine, len));
409     }
410     return lines;
411 }
412
413 int Tools::nextLine(const string &data) {
414     int l = data.size();
415     for (int i = 0; i < l; i++) {
416         if (data[i] == '\n')
417             return i + 1;
418     }
419     return l;
420 }
421
422 string Tools::caseFormat(string text) {
423     vector<string> lines = Tools::splitLines(text);
424     string res;
425     int nlines = lines.size();
426     for (int i = 0; i < nlines; i++)
427         res += ">" + lines[i] + '\n';
428     return res;
429 }
430
431 bool Tools::parseLine(const string &text, string &name, string &data) {
432     size_t poseq;
433     if ((poseq = text.find('=')) != string::npos) {
434         name = normalizeTag(text.substr(0, poseq + 1));
435         data = text.substr(poseq + 1);
436         return true;
437     }
438     name = "";
439     data = text;
440     return false;
441 }
442
443 string Tools::toLower(const string &text) {
444     string res = text;
445     int len = res.size();
446     for (int i = 0; i < len; i++)
447         res[i] = tolower(res[i]);
448     return res;
449 }
450
451 string Tools::normalizeTag(const string &text) {
452     string res;
453     int len = text.size();
454     for (int i = 0; i < len; i++) {
455         char c = text[i];
456         if (isalpha(c) || c == '=')
457             res += tolower(c);
458     }
459     return res;
460 }
461
462 string Tools::trimRight(const string &text) {
463     int len = text.size();
464     int end = -1;
465     for (int i = len - 1; i >= 0; i--) {
466         if (!isspace(text[i])) {
467             end = i;
468             break;
469         }
470     }
471     return text.substr(0, end + 1);
472 }
473
474 string Tools::trim(const string &text) {
475     int len = text.size();
476     int begin = len;
477     int end = -1;
478     for (int i = 0; i < len; i++) {
479         char c = text[i];
480         if (!isspace(c)) {
481             begin = i;
482             break;
483         }
484     }
485     for (int i = len - 1; i >= 0; i--) {
486         char c = text[i];
487         if (!isspace(c)) {
488             end = i;
489             break;
490         }
491     }
492     if (begin <= end)

```



```

493     return text.substr(begin, (end - begin) + 1);
494     return "";
495 }
496
497 void Tools::fdblock(int fd, bool set) {
498     int flags;
499     if ((flags = fcntl(fd, F_GETFL, 0)) < 0) {
500         return;
501     }
502     if (set && (flags | O_NONBLOCK) == flags)
503         flags ^= O_NONBLOCK;
504     else
505         flags |= O_NONBLOCK;
506     fcntl(fd, F_SETFL, flags);
507 }
508
509 bool Tools::convert2(const string& str, double &data){
510     if ( str == ".") {
511         return false;
512     }
513     stringstream conv(str);
514     conv >> data;
515     return conv.eof();
516 }
517
518 bool Tools::convert2(const string& str, long int &data){
519     stringstream conv(str);
520     conv >> data;
521     return conv.eof();
522 }
523 const char* Tools::getenv(const char* name, const char* defaultvalue) {
524     const char* value = ::getenv(name);
525     if ( value == NULL ) {
526         value = defaultvalue;
527         printf("Warning: using default value '%s' for '%s'\n", defaultvalue, name);
528     }
529     return value; // Fixes bug found by Peter Svec
530 }
531
532 double Tools::getenv(const char* name, double defaultvalue) {
533     const char* svalue = ::getenv(name);
534     double value = defaultvalue;
535     if ( svalue != NULL ) {
536         Tools::convert2(svalue, value);
537     } else {
538         printf("Warning: using default value '%lf' for '%s'\n", defaultvalue, name);
539     }
540     return value;
541 }
542
543 /**
544  * Class Stop Definitions
545  */
546
547 void Stop::setTERMRequested() {
548     TERMRequested = true;
549 }
550
551 bool Stop::isTERMRequested() {
552     return TERMRequested;
553 }
554
555 /**
556  * Class Timer Definitions
557  */
558
559 void Timer::start() {
560     startTime = time(NULL);
561 }
562
563 int Timer::elapsedTime() {
564     return time(NULL) - startTime;
565 }
566
567 /**
568  * Class Stop Definitions
569  */
570
571 void I18n::init(){
572 }
573
574 const char *I18n::get_string(const char *s){
575     return s;
576 }
577
578 /**
579  * Class NumbersOutput Definitions
580  */
581
582 // Struct Number
583 bool NumbersOutput::Number::set(const string& str){
584     isInteger=Tools::convert2(str, integer);
585     if(!isInteger){
586         return Tools::convert2(str, Cientific);
587     }
588     return true;
589 }
590
591 bool NumbersOutput::Number::operator==(const Number &o)const{
592     if(isInteger)
593         return o.isInteger && integer == o.integer;
594     if(o.isInteger)
595         return Cientific != 0?fabs((Cientific - o.integer) / Cientific) < 0.0001 : o.integer == 0;
596     else
597         return Cientific != 0?fabs((Cientific - o.Cientific) / Cientific) < 0.0001 : fabs(o.Cientific) < 0.0001;
598 }
599
600 bool NumbersOutput::Number::operator!=(const Number &o)const{
601     return !((*this)==o);
602 }
603
604 NumbersOutput::Number::operator string() const{
605     char buf[100];
606     if(isInteger){
607         sprintf(buf, "%ld", integer);
608     } else {
609         sprintf(buf, "%10.5lf", Cientific);
610     }
611     return buf;
612 }
613
614 }
615

```

```

616
617 bool NumbersOutput::isNum(char c){
618     if(isdigit(c)) return true;
619     return c=='+' || c=='-' || c=='.' || c=='e' || c=='E';
620 }
621
622 bool NumbersOutput::isNumStart(char c){
623     if(isdigit(c)) return true;
624     return c=='+' || c=='-' || c=='.';
625 }
626
627 bool NumbersOutput::calcStartWithAsterisk(){
628     int l=text.size();
629     for(int i=0; i<l; i++){
630         char c=text[i];
631         if(isspace(c)) continue;
632         if(c=='*'){
633             cleanText = text.substr(i+1,text.size()-(i+1));
634             return true;
635         }else{
636             cleanText = text.substr(i,text.size()-i);
637             return false;
638         }
639     }
640     return false;
641 }
642
643 NumbersOutput::NumbersOutput(const string &text):OutputChecker(text){
644     int l=text.size();
645     string str;
646     Number number;
647     for(int i=0; i<l; i++){
648         char c=text[i];
649         if((isNum(c) && str.size()>0) || (isNumStart(c) && str.size()==0)){
650             str+=c;
651         }else if(str.size()>0){
652             if(isNumStart(str[0]) && number.set(str)) numbers.push_back(number);
653             str="";
654         }
655     }
656     if(str.size()>0){
657         if(isNumStart(str[0]) && number.set(str)) numbers.push_back(number);
658     }
659     startWithAsterisk=calcStartWithAsterisk();
660 }
661
662 string NumbersOutput::studentOutputExpected(){
663     return cleanText;
664 }
665
666 bool NumbersOutput::operator==(const NumbersOutput& o)const{
667     size_t l=numbers.size();
668     if( o.numbers.size() < l ) return false;
669     int offset = 0;
670     if(startWithAsterisk)
671         offset = o.numbers.size()-l;
672     for(size_t i = 0; i < l; i++)
673         if(numbers[i] != o.numbers[offset+i])
674             return false;
675     return true;
676 }
677
678 bool NumbersOutput::match(const string& output){
679     NumbersOutput temp(output);
680     return operator==(temp);
681 }
682
683 OutputChecker* NumbersOutput::clone(){
684     return new NumbersOutput(outputExpected());
685 }
686
687 bool NumbersOutput::typeMatch(const string& text){
688     int l=text.size();
689     string str;
690     Number number;
691     for(int i=0; i<l; i++){
692         char c=text[i];
693         // Skip spaces/CR/LF... and *
694         if(!isspace(c) && c!='*') {
695             str += c;
696         }else if(str.size()>0) {
697             if (!isNumStart(str[0])||
698                 !number.set(str)) return false;
699             str="";
700         }
701     }
702     if(str.size()>0){
703         if(!isNumStart(str[0])||!number.set(str)) return false;
704     }
705     return true;
706 }
707
708 string NumbersOutput::type(){
709     return "numbers";
710 }
711
712 NumbersOutput::operator string () const{
713     string ret="[";
714     int l=numbers.size();
715     for(int i=0; i<l; i++){
716         ret += i > 0 ? " " : "";
717         ret += numbers[i];
718     }
719     ret += " ]";
720     return ret;
721 }
722
723 /**
724  * Class TextOutput Definitions
725  */
726
727 bool TextOutput::isAlpha(char c){
728     if ( isalnum(c) ) return true;
729     return c < 0;
730 }
731
732 TextOutput::TextOutput(const string &text):OutputChecker(text){
733     size_t l = text.size();
734     string token;
735     for(size_t i = 0; i < l; i++){
736         char c = text[i];
737         if( isAlpha(c) ){
738             token += c;

```

```

739         }else if(token.size() > 0){
740             tokens.push_back(Tools::toLower(token));
741             token="";
742         }
743     }
744     if(token.size()>0){
745         tokens.push_back(Tools::toLower(token));
746     }
747 }
748
749 bool TextOutput::operator==(const TextOutput& o) {
750     size_t l = tokens.size();
751     if (o.tokens.size() < l) return false;
752     int offset = o.tokens.size() - l;
753     for (size_t i = 0; i < l; i++)
754         if (tokens[i] != o.tokens[ offset + i ])
755             return false;
756     return true;
757 }
758
759 bool TextOutput::match(const string& output) {
760     TextOutput temp(output);
761     return operator== (temp);
762 }
763
764 OutputChecker* TextOutput::clone() {
765     return new TextOutput(outputExpected());
766 }
767
768 bool TextOutput::typeMatch(const string& text) {
769     return true;
770 }
771
772 string TextOutput::type(){
773     return "text";
774 }
775
776 /**
777  * Class ExactTextOutput Definitions
778  */
779
780 bool ExactTextOutput::isAlpha(char c){
781     if(isalnum(c)) return true;
782     return c < 0;
783 }
784
785 ExactTextOutput::ExactTextOutput(const string &text):OutputChecker(text){
786     string clean = Tools::trim(text);
787     if(clean.size() > 2 && clean[0] == '*') {
788         startWithAsterix = true;
789         cleanText = clean.substr(2, clean.size() - 3);
790     }else{
791         startWithAsterix =false;
792         cleanText=clean.substr(1,clean.size()-2);
793     }
794 }
795
796 string ExactTextOutput::studentOutputExpected(){
797     return cleanText;
798 }
799
800 bool ExactTextOutput::operator==(const ExactTextOutput& o){
801     return match(o.text);
802 }
803
804 bool ExactTextOutput::match(const string& output){
805     if(cleanText.size()==0 && output.size()==0) return true;
806     string clean;
807     // Clean output if text last char is alpha
808     if(cleanText.size()>0 && isAlpha(cleanText[cleanText.size()-1])){
809         clean=Tools::trimRight(output);
810     }else{
811         clean=output;
812     }
813     if(startWithAsterix){
814         size_t start=clean.size()-cleanText.size();
815         return cleanText.size()<=clean.size() &&
816             cleanText == clean.substr(start,cleanText.size());
817     }
818     else
819         return cleanText==clean;
820 }
821
822 OutputChecker* ExactTextOutput::clone(){
823     return new ExactTextOutput(outputExpected());
824 }
825
826 bool ExactTextOutput::typeMatch(const string& text){
827     string clean=Tools::trim(text);
828     return (clean.size()>1 && clean[0]==' ' && clean[clean.size()-1]==' ')
829         ||(clean.size()>3 && clean[0]=='*' && clean[1]==' ' && clean[clean.size()-1]==' ');
830 }
831
832 string ExactTextOutput::type(){
833     return "exact text";
834 }
835
836 /**
837  * Class RegularExpressionOutput Definitions
838  */
839
840 RegularExpressionOutput::RegularExpressionOutput(const string &text, const string &actualCaseDescription):OutputChecker(text) {
841
842     errorCase = actualCaseDescription;
843     size_t pos = 1;
844     flagI = false;
845     flagM = false;
846     string clean = Tools::trim(text);
847
848     while (clean[pos] != '/' && pos < clean.size()) {
849         pos++;
850     }
851     cleanText = clean.substr(1,pos-1);
852     if (pos + 1 != clean.size()) {
853         pos = pos + 1;
854
855         // Flag search
856         while (pos < clean.size()) {
857
858             switch (clean[pos]) {
859                 case 'i':
860                     flagI=true;
861                     break;

```

```

862         case 'm':
863             flagM=true;
864             break;
865         default:
866             Evaluation* p_ErrorTest = Evaluation::getSingleton();
867             char wrongFlag = clean[pos];
868             string flagCatch;
869             stringstream ss;
870             ss << wrongFlag;
871             ss >> flagCatch;
872             string errorType = string("Flag Error in case ") + string(errorCase) + string(", found a ") + string(flagCatch) + string(" used as a
873             const char* flagError = errorType.c_str();
874             p_ErrorTest->addFatalError(flagError);
875             p_ErrorTest->outputEvaluation();
876             abort();
877         }
878         pos++;
879     }
880 }
881
882 // Regular Expression compilation (with flags in mind) and comparison with the input and output evaluation
883 bool RegularExpressionOutput::match (const string& output) {
884     reti=-1;
885     const char * in = cleanText.c_str();
886     // Use POSIX-C regex.h
887     // Flag compilation
888     if (flagI || flagM) {
889         if (flagM && flagI) {
890             reti = regcomp(&expression, in, REG_EXTENDED | REG_NEWLINE | REG_ICASE);
891         } else if (flagM) {
892             reti = regcomp(&expression, in, REG_EXTENDED | REG_NEWLINE);
893         } else {
894             reti = regcomp(&expression, in, REG_EXTENDED | REG_ICASE);
895         }
896     }
897     // No flag compilation
898     } else {
899         reti = regcomp(&expression, in, REG_EXTENDED);
900     }
901     if (reti == 0) { // Compilation was successful
902         const char * out = output.c_str();
903         reti = regexec(&expression, out, 0, NULL, 0);
904         if (reti == 0) { // Match
905             return true;
906         } else if (reti == REG_NOMATCH) { // No match
907             return false;
908         } else { // Memory Error
909             Evaluation* p_ErrorTest = Evaluation::getSingleton();
910             string errorType = string("Out of memory error, during matching case ") + string(errorCase);
911             const char* flagError = errorType.c_str();
912             p_ErrorTest->addFatalError(flagError);
913             p_ErrorTest->outputEvaluation();
914             abort();
915         }
916     } else { // Compilation error
917         size_t length = regerror(reti, &expression, NULL, 0);
918         char* bff = new char[length + 1];
919         (void) regerror(reti, &expression, bff, length);
920         Evaluation* p_ErrorTest = Evaluation::getSingleton();
921         string errorType = string("Regular Expression compilation error")+string(" in case: ") + string(errorCase) +string("\n")+ string(bff);
922         const char* flagError = errorType.c_str();
923         p_ErrorTest->addFatalError(flagError);
924         p_ErrorTest->outputEvaluation();
925         abort();
926         return false;
927     }
928 }
929
930 // Returns the expression without flags nor '/'
931 string RegularExpressionOutput::studentOutputExpected() {return cleanText;}
932
933 OutputChecker* RegularExpressionOutput::clone() {
934     return new RegularExpressionOutput(outputExpected(), errorCase);
935 }
936
937 // Tests if it's a regular expression. A regular expressions should be between ././
938 bool RegularExpressionOutput::typeMatch(const string& text) {
939     string clean=Tools::trim(text);
940     if (clean.size() > 2 && clean[0] == '/') {
941         for (size_t i = 1; i < clean.size(); i++) {
942             if (clean[i] == '/') {
943                 return true;
944             }
945         }
946     }
947     return false;
948 }
949
950 string RegularExpressionOutput::type() {
951     return "regular expression";
952 }
953
954 /**
955  * Class Case Definitions
956  * Case represents cases
957  */
958 Case::Case() {
959     reset();
960 }
961
962 void Case::reset() {
963     input = "";
964     output.clear();
965     caseDescription = "";
966     gradeReduction = std::numeric_limits<float>::min();
967     failMessage = "";
968     programToRun = "";
969     programArgs = "";
970     variation = "";
971     expectedExitCode = std::numeric_limits<int>::min();
972 }
973
974 void Case::addInput(string s) {
975     input += s;
976 }
977
978 string Case::getInput() {

```

```

985     return input;
986 }
987
988 void Case::addOutput(string o) {
989     output.push_back(o);
990 }
991
992 const vector< string > & Case::getOutput() {
993     return output;
994 }
995
996 void Case::setFailMessage(const string &s) {
997     failMessage = s;
998 }
999
1000 string Case::getFailMessage() {
1001     return failMessage;
1002 }
1003 void Case::setCaseDescription(const string &s) {
1004     caseDescription = s;
1005 }
1006
1007 string Case::getCaseDescription() {
1008     return caseDescription;
1009 }
1010 void Case::setGradeReduction(float g) {
1011     gradeReduction = g;
1012 }
1013
1014 float Case::getGradeReduction() {
1015     return gradeReduction;
1016 }
1017
1018 void Case::setExpectedExitCode(int e) {
1019     expectedExitCode = e;
1020 }
1021
1022 int Case::getExpectedExitCode() {
1023     return expectedExitCode;
1024 }
1025 void Case::setProgramToRun(const string &s) {
1026     programToRun = s;
1027 }
1028
1029 string Case::getProgramToRun() {
1030     return programToRun;
1031 }
1032
1033 void Case::setProgramArgs(const string &s) {
1034     programArgs = s;
1035 }
1036
1037 string Case::getProgramArgs() {
1038     return programArgs;
1039 }
1040
1041 void Case::setVariation(const string &s) {
1042     variation = Tools::toLower(Tools::trim(s));
1043 }
1044
1045 string Case::getVariation() {
1046     return variation;
1047 }
1048
1049 void Case::setCheckLeak(bool cl) {
1050     this->checkLeak = cl;
1051 }
1052
1053 bool Case::getCheckLeak() {
1054     return checkLeak;
1055 }
1056
1057 /**
1058  * Class TestCase Definitions
1059  * TestCase represents cases of test
1060  */
1061
1062 void TestCase::cutOutputTooLarge(string &output) {
1063     if (output.size() > MAXOUTPUT) {
1064         outputTooLarge = true;
1065         output.erase(0, output.size() - MAXOUTPUT);
1066     }
1067 }
1068
1069 void TestCase::removeZerosFromVector(char buf[], int len) {
1070     for (int i = 0; i < len-1; i++)
1071         if (buf[i] == 0)
1072             buf[i] = 32; // Troca por espacos
1073 }
1074
1075 void TestCase::readWrite(int fdread, int fdwrite) {
1076     const int MAX = 1024*10;
1077     // Buffer size to read
1078     const int POLLREAD = POLLIN | POLLPRI;
1079     // Poll to read from program
1080     struct pollfd devices[2];
1081     devices[0].fd = fdread;
1082     devices[1].fd = fdwrite;
1083     char buf[MAX];
1084     devices[0].events = POLLREAD;
1085     devices[1].events = POLLOUT;
1086     int res = poll(devices, programInput.size()-1, 0);
1087     if (res == -1) // Error
1088         return;
1089     if (res == 0) // Nothing to do
1090         return;
1091     if (devices[0].revents & POLLREAD) { // Read program output
1092         int readed = read(fdread, buf, MAX);
1093         if (readed > 0) {
1094             sizeReaded += readed;
1095             if (programInput.size() > 1) {
1096                 programOutputBefore += string(buf, readed);
1097                 cutOutputTooLarge(programOutputBefore);
1098             } else {
1099                 programOutputAfter += string(buf, readed);
1100                 cutOutputTooLarge(programOutputAfter);
1101             }
1102         }
1103     }
1104     if (programInput.size() > 0 && devices[1].revents & POLLOUT) { // Write to program
1105         int written = write(fdwrite, programInput.c_str(), Tools::nextLine(
1106             programInput));
1107         if (written > 0) {

```

```

1108     programInput.erase(0, written);
1109 }
1110 if(programInput.size()==0){
1111     close(fdwrite);
1112 }
1113 }
1114 }
1115
1116 void TestCase::addOutput(const string &o, const string &actualCaseDescription){
1117 // actualCaseDescription, used to get current test name for Output recognition
1118 if(ExactTextOutput::typeMatch(o))
1119     this->output.push_back(new ExactTextOutput(o));
1120 else if (RegularExpressionOutput::typeMatch(o))
1121     this->output.push_back(new RegularExpressionOutput(o, actualCaseDescription));
1122 else if(NumbersOutput::typeMatch(o))
1123     this->output.push_back(new NumbersOutput(o));
1124 else
1125     this->output.push_back(new TextOutput(o));
1126 }
1127
1128 void TestCase::setEnvironment(const char **environment) {
1129     envv = environment;
1130 }
1131
1132 void TestCase::setDefaultCommand() {
1133     command = "./vpl_test";
1134     argv = new const char*[2];
1135     argv[0] = command;
1136     argv[1] = NULL;
1137 }
1138
1139 TestCase::TestCase(const TestCase &o) {
1140     id=o.id;
1141     correctOutput=o.correctOutput;
1142     hasLeak=o.hasLeak;
1143     checkLeak=o.checkLeak;
1144     correctExitCode = o.correctExitCode;
1145     outputTooLarge=o.outputTooLarge;
1146     programTimeout=o.programTimeout;
1147     executionError=o.executionError;
1148     strcpy(executionErrorReason,o.executionErrorReason);
1149     sizeReaded=o.sizeReaded;
1150     input=o.input;
1151     caseDescription=o.caseDescription;
1152     gradeReduction=o.gradeReduction;
1153     expectedExitCode = o.expectedExitCode;
1154     exitCode = o.exitCode;
1155     failMessage=o.failMessage;
1156     programToRun=o.programToRun;
1157     programArgs=o.programArgs;
1158     gradeReductionApplied=o.gradeReductionApplied;
1159     programOutputBefore=o.programOutputBefore;
1160     programOutputAfter=o.programOutputAfter;
1161     programInput=o.programInput;
1162     for(size_t i = 0; i < o.output.size(); i++){
1163         output.push_back(o.output[i]->clone());
1164     }
1165     setDefaultCommand();
1166 }
1167
1168 TestCase& TestCase::operator=(const TestCase &o) {
1169     id=o.id;
1170     correctOutput=o.correctOutput;
1171     hasLeak=o.hasLeak;
1172     checkLeak=o.checkLeak;
1173     correctExitCode = o.correctExitCode;
1174     outputTooLarge=o.outputTooLarge;
1175     programTimeout=o.programTimeout;
1176     executionError=o.executionError;
1177     strcpy(executionErrorReason,o.executionErrorReason);
1178     sizeReaded=o.sizeReaded;
1179     input=o.input;
1180     caseDescription=o.caseDescription;
1181     gradeReduction=o.gradeReduction;
1182     failMessage=o.failMessage;
1183     programToRun=o.programToRun;
1184     programArgs=o.programArgs;
1185     expectedExitCode = o.expectedExitCode;
1186     exitCode = o.exitCode;
1187     gradeReductionApplied=o.gradeReductionApplied;
1188     programOutputBefore=o.programOutputBefore;
1189     programOutputAfter=o.programOutputAfter;
1190     programInput=o.programInput;
1191     for(size_t i=0; i<o.output.size(); i++)
1192         delete output[i];
1193     output.clear();
1194     for(size_t i=0; i<o.output.size(); i++){
1195         output.push_back(o.output[i]->clone());
1196     }
1197     return *this;
1198 }
1199
1200 TestCase::~TestCase() {
1201     for(size_t i = 0; i < output.size(); i++)
1202         delete output[i];
1203 }
1204
1205 TestCase::TestCase(int id, const string &input, const vector<string> &output,
1206     const string &caseDescription, const float gradeReduction,
1207     string failMessage, string programToRun, string programArgs, int expectedExitCode, bool cl) {
1208     this->id = id;
1209     this->input = input;
1210     for(size_t i = 0; i < output.size(); i++){
1211         addOutput(output[i], caseDescription);
1212     }
1213     this->caseDescription = caseDescription;
1214     this->gradeReduction = gradeReduction;
1215     this->expectedExitCode = expectedExitCode;
1216     this->programToRun = programToRun;
1217     this->programArgs = programArgs;
1218     this->failMessage = failMessage;
1219     this->checkLeak = cl;
1220     exitCode = std::numeric_limits<int>::min();
1221     outputTooLarge = false;
1222     programTimeout = false;
1223     executionError = false;
1224     correctOutput = false;
1225     hasLeak = false;
1226     correctExitCode = false;
1227     sizeReaded = 0;
1228     gradeReductionApplied = 0;
1229     strcpy(executionErrorReason, "");
1230     setDefaultCommand();
1231 }

```

```

1231 }
1232
1233 bool TestCase::isCorrectResult() {
1234     bool correct = correctOutput &&
1235         ! programTimeout &&
1236         ! outputTooLarge &&
1237         ! executionError;
1238     return correct || (isExitCodeTested() && correctExitCode);
1239 }
1240
1241 bool TestCase::hasMemoryLeak() {
1242     return (checkLeak && hasLeak);
1243 }
1244
1245 bool TestCase::isExitCodeTested() {
1246     return expectedExitCode != std::numeric_limits<int>::min();
1247 }
1248
1249 float TestCase::getGradeReduction() {
1250     return gradeReduction;
1251 }
1252
1253 void TestCase::setGradeReductionApplied(float r) {
1254     gradeReductionApplied=r;
1255 }
1256
1257 float TestCase::getGradeReductionApplied() {
1258     return gradeReductionApplied;
1259 }
1260
1261 string TestCase::getCaseDescription(){
1262     return caseDescription;
1263 }
1264
1265 string TestCase::getCommentTitle(bool withGradeReduction=false) {
1266     char buf[100];
1267     string ret;
1268     sprintf(buf, "Test %d", id);
1269     ret = buf;
1270     if (caseDescription.size() > 0) {
1271         ret += " " + caseDescription;
1272     }
1273     if(withGradeReduction && getGradeReductionApplied(>0){
1274         sprintf(buf, "(%.3F)", -getGradeReductionApplied());
1275         ret += buf;
1276     }
1277     ret += '\n';
1278     return ret;
1279 }
1280
1281 string TestCase::getComment() {
1282     string ret;
1283     if (checkLeak && hasLeak) {
1284         ret += "[Memory leak check failed]\n";
1285     }
1286     if (isCorrectResult() && !(checkLeak && hasLeak)) {
1287         return ret;
1288     }
1289     char buf[100];
1290     if(output.size()==0){
1291         ret += "Configuration error in the test case: the output is not defined";
1292     }
1293     if (programTimeout) {
1294         ret += "Program timeout\n";
1295     }
1296     if (outputTooLarge) {
1297         sprintf(buf, "Program output too large (%dKb)\n", sizeReaded / 1024);
1298         ret += buf;
1299     }
1300     if (executionError) {
1301         ret += executionErrorReason + string("\n");
1302     }
1303     if (isExitCodeTested() && ! correctExitCode) {
1304         char buf[250];
1305         sprintf(buf, "Incorrect exit code. Expected %d, found %d\n", expectedExitCode, exitCode);
1306         ret += buf;
1307     }
1308     if (!correctOutput || (checkLeak && hasLeak)) {
1309         if (failMessage.size()) {
1310             ret += correctOutput ? "[Correct program output]\n" : "[Incorrect program output]\n";
1311             ret += failMessage + "\n";
1312         } else {
1313             ret += correctOutput ? "[Correct program output]\n" : "[Incorrect program output]\n";
1314             ret += " --- Input ---\n";
1315             ret += Tools::caseFormat(input);
1316             ret += "\n --- Program output ---\n";
1317             ret += Tools::caseFormat(programOutputBefore + programOutputAfter);
1318             if(output.size(>0){
1319                 ret += "\n --- Expected output (" +output[0]->type()+"---\n";
1320                 ret += Tools::caseFormat(output[0]->studentOutputExpected());
1321             }
1322         }
1323     }
1324     return ret;
1325 }
1326
1327 void TestCase::splitArgs(string programArgs, bool checkLeakOnly) {
1328     const char* original_command = (checkLeakOnly ? argv[1] : command);
1329     int offsetParams = (checkLeakOnly ? 1 : 0);
1330
1331     int l = programArgs.size();
1332     int nargs = 1 + offsetParams;
1333     char *buf = new char[programArgs.size() + 1];
1334     strcpy(buf, programArgs.c_str());
1335
1336     delete argv;
1337     argv = (const char **) new char*[programArgs.size() + 1 + offsetParams];
1338
1339     if (checkLeakOnly) {
1340         argv[0] = "/usr/bin/valgrind";
1341         argv[1] = original_command;
1342     } else {
1343         argv[0] = original_command;
1344     }
1345     bool inArg = false;
1346     char separator = ' ';
1347     for(int i=0; i < l; i++) { // TODO improve
1348         if ( ! inArg ) {
1349             if ( buf[i] == ' ' ) {
1350                 buf[i] = '\0';
1351                 continue;
1352             } else if ( buf[i] == '\\' ) {
1353                 argv[nargs++] = buf + i + 1;

```

```

1354     separator = '\\';
1355     } else if ( buf[i] == ' ' ) {
1356         argv[nargs++] = buf + i + 1;
1357         separator = '\\';
1358     } else if ( buf[i] != '\\0' ) {
1359         argv[nargs++] = buf + i;
1360         separator = ' ';
1361     }
1362     inArg = true;
1363     } else {
1364         if ( buf[i] == separator ) {
1365             buf[i] = '\\0';
1366             separator = '\\';
1367             inArg = false;
1368         }
1369     }
1370 }
1371 argv[nargs] = NULL;
1372 }
1373
1374
1375 void TestCase::runTest(time_t timeout) {
1376     if (checkLeak) runTestExec(timeout, true);
1377     runTestExec(timeout, false);
1378 }
1379
1380 void TestCase::runTestExec(time_t timeout, bool checkLeakOnly) { //timeout in seconds
1381     time_t start = time(NULL);
1382     int pp1[2]; // Send data
1383     int pp2[2]; // Receive data
1384     if (pipe(pp1) == -1 || pipe(pp2) == -1) {
1385         executionError = true;
1386         sprintf(executionErrorReason, "Internal error: pipe error (%s)",
1387                 strerror(errno));
1388         return;
1389     }
1390     command = "./vpl_test";
1391     if ( programToRun > "" && programToRun.size() < 512 ) {
1392         command = programToRun.c_str();
1393     }
1394     if ( ! Tools::existFile(command) ){
1395         executionError = true;
1396         sprintf(executionErrorReason, "Execution file not found '%s'", command);
1397         return;
1398     }
1399
1400     // Updates command and argv if checkLeakOnly
1401     if (checkLeakOnly) {
1402         delete argv;
1403         argv = new const char*[3];
1404         argv[0] = "/usr/bin/valgrind";
1405         argv[1] = command; // First get the original command
1406         argv[2] = NULL;
1407         command = "/usr/bin/valgrind"; // Update command
1408     } else {
1409         delete argv;
1410         argv = new const char*[2];
1411         argv[0] = command;
1412         argv[1] = NULL;
1413     }
1414
1415     pid_t pid;
1416     if ( programArgs.size() > 0 ) {
1417         splitArgs(programArgs, checkLeakOnly);
1418     }
1419
1420     if ((pid = fork()) == 0) {
1421         // Execute
1422         close(pp1[1]);
1423         dup2(pp1[0], STDIN_FILENO);
1424         close(pp2[0]);
1425         dup2(pp2[1], STDOUT_FILENO);
1426         dup2(STDOUT_FILENO, STDERR_FILENO);
1427         setpgpr();
1428         execve(command, (char * const *) argv, (char * const *) envv);
1429         perror("Internal error, execve fails");
1430         abort(); //end of child
1431     }
1432     if (pid == -1) {
1433         executionError = true;
1434         sprintf(executionErrorReason, "Internal error: fork error (%s)",
1435                 strerror(errno));
1436         return;
1437     }
1438     close(pp1[0]);
1439     close(pp2[1]);
1440     int fdwrite = pp1[1];
1441     int fdread = pp2[0];
1442     Tools::fdblock(fdwrite, false);
1443     Tools::fdblock(fdread, false);
1444     programInput = input;
1445     if(programInput.size()==0){ // No input
1446         close(fdwrite);
1447     }
1448     programOutputBefore = "";
1449     programOutputAfter = "";
1450     pid_t pidr;
1451     int status;
1452     exitCode = std::numeric_limits<int>::min();
1453     while ((pidr = waitpid(pid, &status, WNOHANG | WUNTRACED)) == 0) {
1454         readWrite(fdread, fdwrite);
1455         usleep(5000);
1456         // TERMSIG or timeout or program output too large?
1457         if (Stop::isTERMRequested() || (time(NULL) - start) >= timeout
1458             || outputTooLarge) {
1459             if ((time(NULL) - start) >= timeout) {
1460                 programTimeout = true;
1461             }
1462             kill(pid, SIGTERM); // Send SIGTERM normal termination
1463             int otherstatus;
1464             usleep(5000);
1465             if (waitpid(pid, &otherstatus, WNOHANG | WUNTRACED) == pid) {
1466                 break;
1467             }
1468             if (kill(pid, SIGQUIT) == 0) { // Kill
1469                 break;
1470             }
1471         }
1472     }
1473     if (pidr == pid) {
1474         if (WIFSIGNALED(status)) {
1475             int signal = WTERMSIG(status);
1476             executionError = true;

```



```

1477         sprintf(executionErrorReason,
1478                 "Program terminated due to \"%s\" (%d)\n", strsignal(
1479                     signal), signal);
1480     }
1481     if (WIFEXITED(status)) {
1482         exitCode = WEXITSTATUS(status);
1483     } else {
1484         executionError = true;
1485         strcpy(executionErrorReason,
1486                 "Program terminated but unknown reason.");
1487     }
1488     } else if (pidr != 0) {
1489         executionError = true;
1490         strcpy(executionErrorReason, "waitpid error");
1491     }
1492     readWrite(fdread, fdwrite);
1493
1494     if (checkLeakOnly)
1495         hasLeak = !matchMemoryLeak(programOutputAfter);
1496     else {
1497         correctExitCode = isExitCodeTested() && expectedExitCode == exitCode;
1498         correctOutput = match(programOutputAfter)
1499             || match(programOutputBefore + programOutputAfter);
1500     }
1501 }
1502
1503 bool TestCase::matchMemoryLeak(string data) {
1504     if (!checkLeak) return true;
1505
1506     const char *out = data.c_str();
1507
1508     regex_t regexp_str;
1509     regcomp(&regexp_str, "All heap blocks were freed -- no leaks are possible", REG_EXTENDED|REG_NOSUB);
1510
1511     int retRegex = regexec(&regexp_str, out, 0, NULL, 0);
1512     return (retRegex == 0); // 0 = Match
1513 }
1514
1515 bool TestCase::match(string data) {
1516     for (size_t i = 0; i < output.size(); i++)
1517         if (output[i] -> match(data))
1518             return true;
1519     return false;
1520 }
1521
1522 /**
1523  * Class Evaluation Definitions
1524  */
1525
1526 Evaluation::Evaluation() {
1527     grade = 0;
1528     ncomments = 0;
1529     nerrors = 0;
1530     nruns = 0;
1531     noGrade = true;
1532 }
1533
1534 Evaluation* Evaluation::getSingleton() {
1535     if (singleton == NULL) {
1536         singleton = new Evaluation();
1537     }
1538     return singleton; // Fixes by Jan Derriks
1539 }
1540
1541 void Evaluation::deleteSingleton(){
1542     if (singleton != NULL) {
1543         delete singleton;
1544         singleton = NULL;
1545     }
1546 }
1547
1548 void Evaluation::addTestCase(Case &caso) {
1549     if ( caso.getVariation().size() && caso.getVariation() != variation ) {
1550         return;
1551     }
1552     testCases.push_back(TestCase(testCases.size() + 1, caso.getInput(), caso.getOutput(),
1553     caso.getCaseDescription(), caso.getGradeReduction(), caso.getFailMessage(),
1554     caso.getProgramToRun(), caso.getProgramArgs(), caso.getExpectedExitCode(), caso.getCheckLeak() ) );
1555 }
1556
1557 void Evaluation::removeLastNL(string &s) {
1558     if (s.size() > 0 && s[s.size() - 1] == '\n') {
1559         s.resize(s.size() - 1);
1560     }
1561 }
1562
1563 bool Evaluation::cutToEndTag(string &value, const string &endTag) {
1564     size_t pos;
1565     if (endTag.size() && (pos = value.find(endTag)) != string::npos) {
1566         value.resize(pos);
1567         return true;
1568     }
1569     return false;
1570 }
1571
1572 void Evaluation::loadTestCases(string fname) {
1573     if (!Tools::existFile(fname)) return;
1574     const char *CASE_TAG = "case=";
1575     const char *INPUT_TAG = "input=";
1576     const char *INPUT_END_TAG = "inputend=";
1577     const char *OUTPUT_TAG = "output=";
1578     const char *OUTPUT_END_TAG = "outputend=";
1579     const char *GRADEREDUCTION_TAG = "gradereduction=";
1580     const char *FAILMESSAGE_TAG = "failmessage=";
1581     const char *PROGRAMTORUN_TAG = "programtorun=";
1582     const char *PROGRAMARGS_TAG = "programarguments=";
1583     const char *EXPECTEDEXITCODE_TAG = "expectedexitcode=";
1584     const char *VARIATION_TAG = "variation=";
1585     const char *HIDE_INPUT_OUTPUT_TAG = "hideinputoutput=";
1586
1587     enum {
1588         regular, ininput, inoutput
1589     } state;
1590     bool inCase = false;
1591     vector<string> lines = Tools::splitLines(Tools::readFile(fname));
1592     remove(fname.c_str());
1593     string inputEnd = "";
1594     string outputEnd = "";
1595     Case caso;
1596
1597     caso.setCheckLeak(CHECK_LEAK_FLAG);
1598
1599     string output = "";

```

```

1600 string tag, value;
1601 /* must be changed from String
1602  * to pair type (regexp o no) and string. */
1603 state = regular;
1604 int nlines = lines.size();
1605 for (int i = 0; i < nlines; i++) {
1606     string &line = lines[i];
1607     Tools::parseline(line, tag, value);
1608     if (state == ininput) {
1609         if (inputEnd.size()) { // Check for end of input.
1610             size_t pos = line.find(inputEnd);
1611             if (pos == string::npos) {
1612                 caso.addInput(line + "\n");
1613             } else {
1614                 cutToEndTag(line, inputEnd);
1615                 caso.addInput(line);
1616                 state = regular;
1617                 continue; // Next line.
1618             }
1619         } else if (tag.size() && (tag == OUTPUT_TAG || tag
1620             == GRADEREDUCTION_TAG || tag == CASE_TAG)) { // New valid tag.
1621             state = regular;
1622             // Go on to process the current tag.
1623         } else {
1624             caso.addInput(line + "\n");
1625             continue; // Next line.
1626         }
1627     } else if (state == inoutput) {
1628         if (outputEnd.size()) { // Check for end of output.
1629             size_t pos = line.find(outputEnd);
1630             if (pos == string::npos) {
1631                 output += line + "\n";
1632             } else {
1633                 cutToEndTag(line, outputEnd);
1634                 output += line;
1635                 caso.addOutput(output);
1636                 output = "";
1637                 state = regular;
1638                 continue; // Next line.
1639             }
1640         } else if (tag.size() && (tag == INPUT_TAG || tag == OUTPUT_TAG
1641             || tag == GRADEREDUCTION_TAG || tag == CASE_TAG)) { // New valid tag.
1642             removeLastNL(output);
1643             caso.addOutput(output);
1644             output = "";
1645             state = regular;
1646         } else {
1647             output += line + "\n";
1648             continue; // Next line.
1649         }
1650     }
1651     if (state == regular && tag.size()) {
1652         if (tag == INPUT_TAG) {
1653             inCase = true;
1654             if (cutToEndTag(value, inputEnd)) {
1655                 caso.addInput(value);
1656             } else {
1657                 state = ininput;
1658                 caso.addInput(value + '\n');
1659             }
1660         } else if (tag == OUTPUT_TAG) {
1661             inCase = true;
1662             if (cutToEndTag(value, outputEnd)) {
1663                 caso.addOutput(value);
1664             } else {
1665                 state = inoutput;
1666                 output = value + '\n';
1667             }
1668         } else if (tag == GRADEREDUCTION_TAG) {
1669             inCase = true;
1670             value = Tools::trim(value);
1671             // A percent value?
1672             if (value.size() > 1 && value[value.size() - 1] == '%') {
1673                 float percent = atof(value.c_str());
1674                 caso.setGradeReduction((grademax-grademin)*percent/100);
1675             } else {
1676                 caso.setGradeReduction( atof(value.c_str()) );
1677             }
1678         } else if (tag == EXPECTEDEXITCODE_TAG) {
1679             caso.setExpectedExitCode( atoi(value.c_str()) );
1680         } else if (tag == PROGRAMTORUN_TAG) {
1681             caso.setProgramToRun(Tools::trim(value));
1682         } else if (tag == PROGRAMARGS_TAG) {
1683             caso.setProgramArgs(Tools::trim(value));
1684         } else if (tag == FAILMESSAGE_TAG) {
1685             caso.setFailMessage(Tools::trim(value));
1686         } else if (tag == VARIATION_TAG) {
1687             caso.setVariation(value);
1688         } else if (tag == INPUT_END_TAG) {
1689             inputEnd = Tools::trim(value);
1690         } else if (tag == OUTPUT_END_TAG) {
1691             outputEnd = Tools::trim(value);
1692         } else if (tag == HIDE_INPUT_OUTPUT_TAG) {
1693             string hide_input_output_str = Tools::trim(value);
1694             bool hideInputOutput = (hide_input_output_str.compare("True") == 0);
1695             if (hideInputOutput && !caso.getFailMessage().size()) caso.setFailMessage( caso.getCasoDescription() );
1696         } else if (tag == CASE_TAG) {
1697             if (inCase) {
1698                 addTestCase(caso);
1699                 caso.reset();
1700             }
1701             inCase = true;
1702             caso.setCaseDescription( Tools::trim(value) );
1703         } else {
1704             if ( line.size() > 0 ) {
1705                 char buf[250];
1706                 sprintf(buf, "Syntax error: unexpected line %d", i+1);
1707                 addFatalError(buf);
1708             }
1709         }
1710     }
1711 }
1712 // TODO review
1713 if (state == inoutput) {
1714     removeLastNL(output);
1715     caso.addOutput(output);
1716 }
1717 if (inCase) { // Last case => save current.
1718     addTestCase(caso);
1719 }
1720 }
1721
1722 bool Evaluation::loadParams() {

```

```

1723     grademin= Tools::getenv("VPL_GRADEMIN", 0.0);
1724     grademax = Tools::getenv("VPL_GRADEMAX", 10);
1725     maxtime = (int) Tools::getenv("VPL_MAXTIME", 20);
1726     variation = Tools::toLower(Tools::trin(Tools::getenv("VPL_VARIATION", "")));
1727     noGrade = grademin >= grademax;
1728     return true;
1729 }
1730
1731 void Evaluation::addFatalError(const char *m) {
1732     float reduction = grademax - grademin;
1733     if (ncomments >= MAXCOMMENTS)
1734         ncomments = MAXCOMMENTS - 1;
1735
1736     snprintf(titles[ncomments], MAXCOMMENTSTITLELENGTH, "%s", m);
1737     snprintf(titlesGR[ncomments], MAXCOMMENTSTITLELENGTH, "%s (%.2f)", m, reduction);
1738     strcpy(comments[ncomments], "");
1739     ncomments ++;
1740     grade = grademin;
1741 }
1742
1743 void Evaluation::runTests() {
1744     if (testCases.size() == 0) {
1745         return;
1746     }
1747     if (maxtime < 0) {
1748         addFatalError("Global timeout");
1749         return;
1750     }
1751     nerrors = 0;
1752     nruns = 0;
1753     bool hasMemoryLeakFlag = false;
1754     grade = grademax;
1755     float defaultGradeReduction = (grademax - grademin) / testCases.size();
1756     int timeout = maxtime / testCases.size();
1757     for (size_t i = 0; i < testCases.size(); i++) {
1758         printf("Testing %lu/%lu : %s\n", (unsigned long) i+1, (unsigned long) testCases.size(), testCases[i].getCaseDescription().c_str());
1759         if (timeout <= 1 || Timer::elapsedTime() >= maxtime) {
1760             grade = grademin;
1761             addFatalError("Global timeout");
1762             return;
1763         }
1764         if (maxtime - Timer::elapsedTime() < timeout) { // Try to run last case
1765             timeout = maxtime - Timer::elapsedTime();
1766         }
1767         testCases[i].runTest(timeout);
1768         nruns++;
1769         if (!testCases[i].isCorrectResult() || testCases[i].hasMemoryLeak()) {
1770             if (Stop::isTERMRequested())
1771                 break;
1772             if (!testCases[i].isCorrectResult()) {
1773                 float gr = testCases[i].getGradeReduction();
1774                 if (gr == std::numeric_limits<float>::min())
1775                     testCases[i].setGradeReductionApplied(defaultGradeReduction);
1776                 else
1777                     testCases[i].setGradeReductionApplied(gr);
1778                 grade -= testCases[i].getGradeReductionApplied();
1779                 if (grade < grademin) {
1780                     grade = grademin;
1781                 }
1782             }
1783             if (testCases[i].hasMemoryLeak())
1784                 hasMemoryLeakFlag = true;
1785             nerrors++;
1786             if (ncomments < MAXCOMMENTS) {
1787                 strncpy(titles[ncomments], testCases[i].getCommentTitle().c_str(),
1788                     MAXCOMMENTSTITLELENGTH);
1789                 strncpy(titlesGR[ncomments], testCases[i].getCommentTitle(true).c_str(),
1790                     MAXCOMMENTSTITLELENGTH);
1791                 strncpy(comments[ncomments], testCases[i].getComment().c_str(),
1792                     MAXCOMMENTSLENGTH);
1793                 ncomments++;
1794             }
1795         }
1796     }
1797
1798     if (hasMemoryLeakFlag) {
1799         grade -= MEMORY_LEAK_REDUCTION;
1800         printf("\n<|--\n");
1801         printf("-Memory leak: -%.2f\n", MEMORY_LEAK_REDUCTION);
1802         printf("-->\n");
1803     }
1804     if (grade < grademin)
1805         grade = grademin;
1806 }
1807
1808 void Evaluation::outputEvaluation() {
1809     const char* stest[] = {"test", "tests"};
1810     if (testCases.size() > 0) {
1811         if (ncomments > 1) {
1812             printf("\n<|--\n");
1813             printf("-Failed tests\n");
1814             for (int i = 0; i < ncomments; i++) {
1815                 printf("%s", titlesGR[i]);
1816             }
1817             printf("-->\n");
1818         }
1819         if (ncomments > 0) {
1820             printf("\n<|--\n");
1821             for (int i = 0; i < ncomments; i++) {
1822                 printf("-%s", titlesGR[i]);
1823                 printf("%s\n", comments[i]);
1824             }
1825             printf("-->\n");
1826         }
1827         int passed = nruns - nerrors;
1828         if (nruns > 0) {
1829             printf("\n<|--\n");
1830             printf("-Summary of tests\n");
1831             printf(">+-----+>\n");
1832             printf(">| %2d %s run/%2d %s passed |>\n",
1833                 nruns, nruns==1?stest[0]:stest[1],
1834                 passed, passed==1?stest[0]:stest[1]); // Taken from Dominique Thiebaut
1835             printf(">+-----+>\n");
1836             printf("\n-->\n");
1837         }
1838         if (!noGrade) {
1839             char buf[100];
1840             sprintf(buf, "%5.2f", grade);
1841             int len = strlen(buf);
1842             if (len > 3 && strcmp(buf + (len - 3), ".00") == 0)
1843                 buf[len - 3] = 0;
1844             printf("\nGrade :>=>%s\n", buf);
1845         }
1846     }

```

```

1846     } else {
1847         printf("<|--\n");
1848         printf("-No test case found\n");
1849         printf("-->\n");
1850     }
1851     fflush(stdout);
1852 }
1853
1854 void nullSignalCatcher(int n) {
1855     //printf("Signal %d\n",n);
1856 }
1857
1858 void signalCatcher(int n) {
1859     //printf("Signal %d\n",n);
1860     if (Stop::isTERMRequested()) {
1861         Evaluation* obj = Evaluation::getSingleton();
1862         obj->outputEvaluation();
1863         abort();
1864     }
1865     Evaluation *obj = Evaluation::getSingleton();
1866     if (n == SIGTERM) {
1867         obj->addFatalError("Global test timeout (TERM signal received)");
1868     } else {
1869         obj->addFatalError("Internal test error");
1870         obj->outputEvaluation();
1871         Stop::setTERMRequested();
1872         abort();
1873     }
1874     alarm(1);
1875 }
1876
1877 void setSignalsCatcher() {
1878     // Removes as many signal controllers as possible
1879     for(int i=0; i<31; i++)
1880         signal(i, nullSignalCatcher);
1881     signal(SIGINT, signalCatcher);
1882     signal(SIGQUIT, signalCatcher);
1883     signal(SIGILL, signalCatcher);
1884     signal(SIGTRAP, signalCatcher);
1885     signal(SIGFPE, signalCatcher);
1886     signal(SIGSEGV, signalCatcher);
1887     signal(SIGALRM, signalCatcher);
1888     signal(SIGTERM, signalCatcher);
1889 }
1890
1891 int main(int argc, char *argv[], const char **env) {
1892     Timer::start();
1893     TestCase::setEnvironment(env);
1894     setSignalsCatcher();
1895     Evaluation* obj = Evaluation::getSingleton();
1896     obj->loadParams();
1897     obj->loadTestCases("evaluate.cases");
1898     obj->runTests();
1899     obj->outputEvaluation();
1900     return EXIT_SUCCESS;
1901 }
1902

```

vpl_test_main.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "pilha.h"
4
5  int main() {
6      int ret, valor;
7      char operacao;
8
9      PilhaDinamica *pilha = criar_pilha();
10
11      scanf(" %c", &operacao);
12      while (operacao != 'S') {
13          if (operacao == 'E') {
14              scanf("%d", &valor);
15              ret = empilhar(pilha, valor);
16              if (ret)
17                  printf("E %d\n", valor);
18              else
19                  printf("Pilha cheia\n");
20          }
21          if (operacao == 'D') {
22              ret = desempilhar(pilha, &valor);
23              if (ret)
24                  printf("D %d\n", valor);
25              else
26                  printf("Pilha vazia\n");
27          }
28          scanf(" %c", &operacao);
29      }
30      liberar_pilha(pilha);
31
32      return 0;
33  }
34

```

pilha.h

```

1  #include "lista.h"
2
3  #ifndef PILHA_DINAMICA
4  #define PILHA_DINAMICA
5
6  typedef struct PilhaDinamica PilhaDinamica;
7  struct PilhaDinamica {
8      LinkedNode *topo;
9  };
10
11  PilhaDinamica *criar_pilha();
12  int empilhar(PilhaDinamica *pilha, int item);
13  int desempilhar(PilhaDinamica *pilha, int *item);
14  void liberar_pilha(PilhaDinamica *pilha);
15
16  #endif

```

lista.h

```
1 #ifndef LISTA_LIGADA
2 #define LISTA_LIGADA
3
4 typedef struct LinkedNode LinkedNode;
5 struct LinkedNode {
6     int data;
7     LinkedNode *next;
8 };
9
10 #endif
```

[VPL](#)[◀ \[EP\] Edição DNA](#)[\[EP\] Fila dinâmica \(hidden\) ▶](#)

Este é o Ambiente Virtual de Aprendizagem da UFABC para apoio ao ensino presencial e semipresencial. Esta plataforma permite que os usuários (educadores/alunos) possam criar cursos, gerenciá-los e participar de maneira colaborativa.

Info

[Conheça a UFABC](#)[Conheça o NTI](#)[Conheça o Netel](#)

Contact us

Av. dos Estados, 5001. Bairro Bangu - Santo André /SP – Brasil. CEP 09210-580.

Follow us



Universidade Federal do ABC - Moodle (2020)

[English \(en\)](#)[English \(en\)](#)[Português - Brasil \(pt_br\)](#)[Get the mobile app](#)