Projecte de ESIN
Normativa i Enunciat

Tardor de 2022

Aquest document és llarg però és imprescindible que el llegiu íntegrament i amb deteniment, àdhuc si sou repetidors, ja que es donen les instruccions i normes que heu de seguir per a que el vostre projecte sigui avaluat positivament. El professorat de l'assignatura donarà per fet que tots els alumnes coneixen el contingut íntegre d'aquest document.

Continguts:

1 Normativa

1. Tal i com s'explica a la Guia Docent, per a assolir els objectius de l'assignatura es considera imprescindible el desenvolupament per part de l'estudiant d'un projecte que requereix algunes hores addicionals de treball personal, apart de les classes de laboratori, on es fa el desenvolupament dels altres exercicis pràctics que permeten familiaritzar-vos amb l'entorn de treball i el llenguatge de programació C++.

2. El projecte es realitzarà en equips de dos estudiants. Si un d'ells abandona, un dels integrants ho haurà de notificar amb la màxima promptitud via e-mail a `jesteve@cs.upc.edu` i, eventualment, continuar el projecte en solitari. D'altra banda, només es permetrà la formació d'equips individuals en casos excepcionals on sigui impossible reunir-se o comunicar-se amb altres estudiants, i s'haurà de justificar mitjançant algun tipus de document.

3. El suport que fareu servir per aquest projecte és el llenguatge de programació C++ (específicament el compilador GNU `g++-9.4.0`) sobre l'entorn Linux del STIC. Això no és obstacle per al desenvolupament previ en PCs o similars. De fet, existeixen compiladors de C++ per a tota classe de plataformes i hauria de ser senzill el trasllat des del vostre equip particular a l'entorn del STIC, especialment si treballeu amb GNU/Linux en el vostre PC.

   Atenció: Existeix la possibilitat de petites incompatibilitats entre alguns compiladors de C++. En tot cas és imprescindible que feu almenys una comprovació final que el programa desenvolupat en PC o similar funciona correctament en l'entorn Linux del STIC (us podeu connectar remotament al servidor `ubiwan.epsevg.upc.edu`).

4. El projecte serà avaluat mitjançant:

   - la seva execució en l'entorn Linux del STIC amb una sèrie de *jocs de prova* i
   - la correcció del disseny, implementació i documentació: les decisions de disseny i la seva justificació, l'eficiència dels algorismes i estructures de dades, la legibilitat, robustesa i estil de programació, etc. Tota la documentació ha d'acompanyar el codi; no heu de lliurar cap documentació en paper.

   Existeixen dos tipus de jocs de prova: públics i privats. Els jocs de prova públics per a que podeu provar el vostre projecte estaran a la vostra disposició amb antelació suficient al Campus Digital (`https://atenea.upc.edu`).

   La nota del projecte es calcula a partir de la nota d'execució (E) i la nota de disseny (D). La nota total és:

   $$P = 0.5E + 0.5D$$

   si ambdues notes parcials (E i D) són majors que 0; $P = 0$ si la nota de disseny és 0.

   El capítol G del *Manual de laboratori d'ESIN* descriu, entre altres coses, les situacions que originen una qualificació de 0 en el disseny (i per tant una qualificació de 0 del projecte). La nota d'execució (E) és 3 punts com a mínim si s'han superat els jocs

de prova públics; en cas contrari, la nota és 0. Els jocs de prova privats poden aportar fins a 7 punts més, en cas que s'hagin superat els jocs de prova públics.

5. La data límit del lliurament final és el 15 de gener de 2023 a les 23h. del vespre. Si un equip no ha lliurat el projecte llavors la nota serà 0. Al Campus Digital (`https://atenea.upc.edu`) es donaran tots els detalls sobre el procediment de lliurament del projecte.

6. No subestimeu el temps que haureu d'esmerçar a cadascun dels aspectes del projecte: disseny, codificació, depuració d'errors, proves, documentació, ...

2 Enunciat del projecte

En aquest projecte heu de desenvolupar una sèrie de classes i mòduls per pertanyen a un petit programa de computació simbòlica, al que anomenem ESINMATH. Si bé no ofereix—ni molt menys—la funcionalitat pròpia de programes com MAPLE, MATHEMATICA o MUPAD, ESINMATH és capaç de realitzar les funcions més bàsiques típiques d'aquests sistemes.

Habitualment ESINMATH s'usarà en mode interactiu: l'usuari entra una comanda com a resposta al *prompt* del sistema, ESINMATH la processa i dóna la resposta requerida, i torna a esperar una nova comanda de l'usuari.

Hi ha quatre tipus de comandes reconeguts per ESINMATH:

1. *expressions* que ESINMATH avalua i el resultat de les quals imprimeix.
2. assignacions de la forma $x := E$, on x és una variable qualsevol i E una expressió.
3. `unassign` x, ens permet desassignar el valor assignat a una variable x.
4. la comanda `byebye` finalitza la sessió amb ESINMATH.

El següent exemple mostra una possible interacció amb el sistema (les respostes del sistema apareixen amb un color de lletra diferent i les comandes entrades per l'usuari apareixen precedits pel *prompt* `'>'`). Per comoditat representarem en aquest document l'operació d'exponenciació mitjançant l'accent circumflex (^).

```
> 2 + 5
7
> -2+8*9
70
> f:=a-b*c
a-b*c
> a:=3
3
> f
3-b*c
> b:= 0
0
> f
3
> 3/4-1/3
5/12
> sqrt(2)
2^(1/2)
```

```
> evalf(%)
1.414213562
> p:= x^2-8*x+3
x^2-8*x+3
> q:= diff(p,x)
2*x-8
> x:= 3
3
> q
-2
> unassign x
x
> q
2*x-8
```

ESINMATH pot gestionar expressions en les que intervinguin nombres enters, racionals i reals en coma flotant, emmagatzemar expressions en variables, aplicar funcions com `sqrt` (arrel quadrada), logaritmes, exponencials, calcular el valor d'una expressió en coma flotant i realitzar simplificacions.

La funció `evalf` avalua l'expressió donada en coma flotant. El símbol `%` és sempre l'última expressió avaluada. L'exemple de les dues últimes línees és important atès que mostra una característica singular d'ESINMATH: una vegada desassignem el valor a la variable x l'avaluació de la variable q té com a resultat l'expressió que se li va assignar originalment, és a dir, el seu valor.

2.1 Sintaxi

Com hem vist en els exemples, les expressions que gestiona ESINMATH involucren a constants, variables, operadors i funcions predefinides.

Les constants numèriques poden ser de tres tipus: un nombre enter, un nombre racional i un nombre real en coma flotant. Per exemple, -77, -77/77, 77.77E77,... Una constant entera s'escriu com una seqüència de dígits del 0 al 9, precedida opcionalment per un signe ('+' o '-'). Una constant real en coma flotant s'escriu com una mantissa seguida opcionalment d'un exponent; entre la mantissa i l'exponent s'escriu la lletra 'E'. L'exponent és una constant entera. La mantissa té una part entera seguida opcionalment per un punt ('.') i una part fraccionària. La part entera és una constant entera; la part fraccionària és una constant entera sense signe. Les constants racionals són el resultat de realitzar el quocient de dues constants enteres.

Les variables s'identifiquen mitjançant un nom. Els noms de les variables estan formats per una o més lletres majúscules o minúscules i el caràcter de subratllat ('_'). No es permetran altres caràcters. El nom d'una variable no pot coincidir amb el nom d'una funció predefinida (`sqrt`, `exp`, `log`, `evalf`) ni de la constant predefinida `e` (la base dels logaritmes naturals, $e = 2.718281\ldots$). A més de les variables definides per l'usuari, ESINMATH reconeix la variable predefinida (`%`).

Els operadors reconeguts per ESINMATH són: la suma (+), la resta (-), el canvi de signe (-), la multiplicació (*), la divisió (/), l'exponenciació (^), l'assignació (:=) i l'operador `unassign`. Aquests dos últims no són part d'expresions, però s'utilitzen, respectivament, en les comandes de la forma $x := E$ i `unassign` x.

Addicionalment es poden utilitzar parèntesis per agrupar subexpressions i canviar la prioritat o associativitat d'aplicació dels operadors.

A continuació es presenta la sintaxi formal de les expressions reconegudes per ESINMATH.

- Qualsevol constant entera, racional o real en coma flotant és una expressió vàlida. També ho és la constant predefinida `e`.
- Qualsevol variable i la variable especial `%` és una expressió vàlida.
- Si E és una expressió vàlida llavors (E) i $-E$ i $+E$ són expressions vàlides.
- Si E_1 i E_2 són expressions vàlides llavors també ho són $E_1 + E_2$, $E_1 - E_2$, $E_1 * E_2$, E_1/E_2 i $E_1\hat{}\,E_2$.
- Si f és una de les funcions predefinides `sqrt`, `exp`, `log` o `evalf` i E és una expressió vàlida llavors $f(E)$ és una expressió vàlida.

2.2 Precedència i associativitat

Per ordre decreixent de precedència els operadors s'ordenen així:

Operadors	Associativitat
^ (exponenciació)	De dreta a esquerra
- (canvi de signe), + (signe positiu)	D'esquerra a dreta
$*$ (multiplicació), / (divisió)	D'esquerra a dreta
+ (suma), - (resta)	D'esquerra a dreta

Quan dos operadors són de la mateixa prioritat s'avaluaran d'esquerra a dreta excepte l'exponenciació que s'avaluarà de dreta a esquerra. Per exemple, la següent expressió

```
a+b*c^d^e/f
```

s'ha d'interpretar com

```
a+((b*(c^(d^e))/f)
```

2.3 Semàntica

Òbviament existeixen expressions que encara que siguin sintàcticament vàlides no tenen sentit (no tenen semàntica) o no poden ser avaluades en cap context. Per exemple, per qualsevol expressió E, $E/0$ no té valor possible. O no podem derivar una expressió E respecte a qualsevol cosa que no sigui una variable. Tampoc podem avaluar el logaritme o l'arrel d'un nombre negatiu. I si tenim una variable "formal" x (una variable que no té assignat cap valor) llavors expressions tals com $x := x + 1$ donen origen a recursions infinites ...

Totes aquestes consideracions s'hauran de tenir en compte quan es defineixin les regles d'avaluació i simplificació d'expressions. Abans, però, descriurem breument com funciona ESINMATH.

1. Un cop llegida l'expressió, ESINMATH descomposa l'expressió en una seqüència de `tokens`; cada `token` és un valor literal (una constant entera, racional o en coma flotant), un identificador (de variable o símbol de funció), un operador (+, *, ...), un parèntesi d'obertura o tancament o una coma.

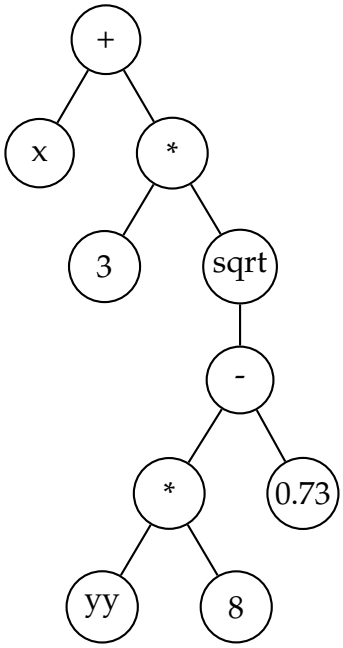
2. A continuació construeix un *arbre d'expressió*, comprovant a la vegada la correctesa sintàctica de l'expressió llegida. Per exemple, si la seqüència de caràcters entrada per l'usuari fos

```
x+3*sqrt((yy*8)-7.3E-1)
```

aquesta es descomposaria en la següent seqüència de `tokens` (després de la fase que s'anomena *anàlisi lèxic*)

x	+	3	*	sqrt	(	(	yy	*	8	)	-	7.3E-1	)

L'expressió es representarà mitjançant el següent *arbre d'expressió*



3. L'algorisme que s'usa per construir l'*arbre d'expressió* és molt similar al que s'usa per convertir una expressió en notació infixa (l'habitual) a l'anomenada notació postfixa o *polaca*. La generació de l'*arbre d'expressió*, incloent la detecció d'errors de sintaxi, constitueix la fase que s'anomena *anàlisi sintàctic*.

4. L'últim pas consisteix en avaluar l'expressió. L'avaluació d'una expressió és una altra expressió i això és el que ESINMATH acabarà imprimint en el canal estàndar de sortida—també emmagatzemarà l'expressió resultant a la variable especial `%`. Aquesta fase en la que es simplifica i s'avalua una expressió s'anomena fase *semàntica*.

2.4 Construcció de l'arbre d'expressió

Com s'ha comentat, l'algorisme que s'usa per construir l'*arbre d'expressió* és molt similar al que s'usa per convertir una expressió en notació infixa (l'habitual) a l'anomenada notació postfixa o *polaca*. És l'anomenat *algorisme de la platja de maniobres* (*Shunting*

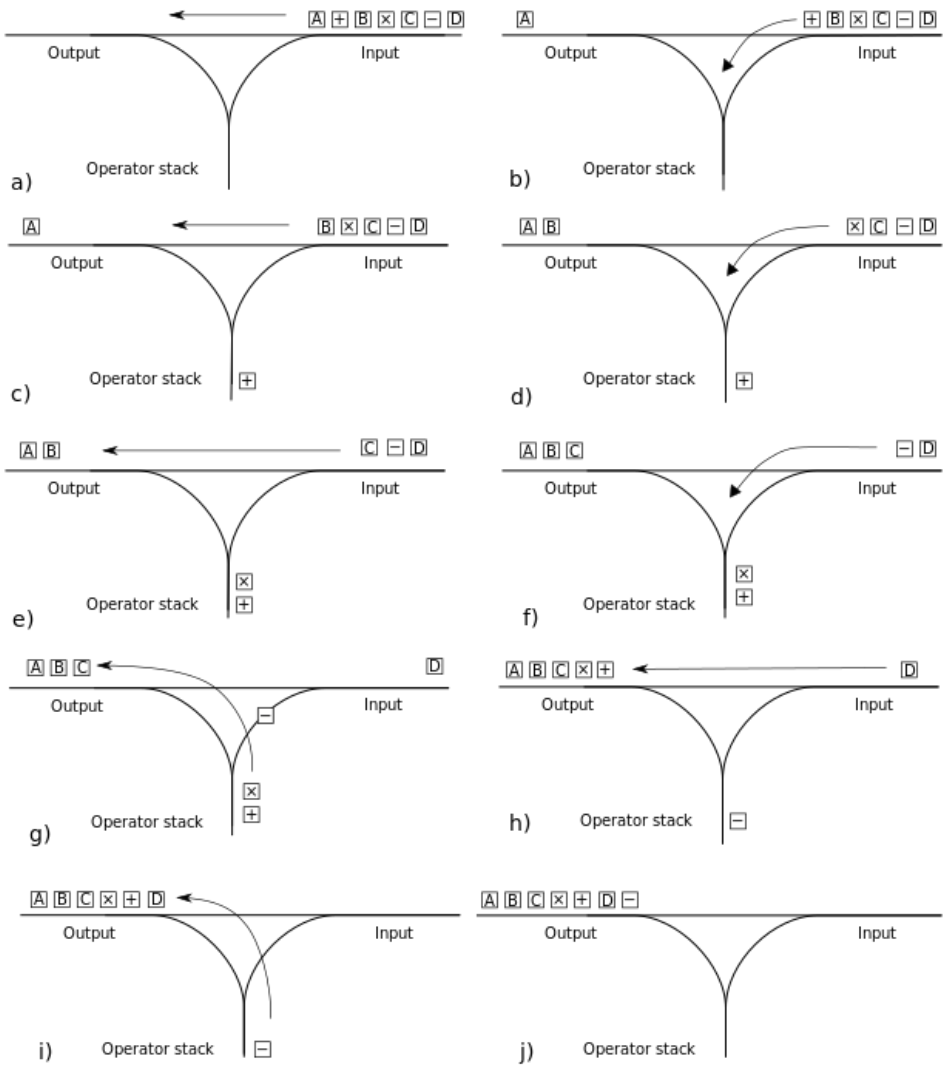


Figura 1: Conversió expressió infixa a postfixa.
By Salix alba - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=10960619

yard algorithm[1]) per la seva similitud a l'operativa ferroviària de moure vagons de tren de mercaderies en una platja de vies (veure figura 1)

Podem implementar-ho usant una pila per guardar els operadors i els parèntesis d'obertura i una pila d'expressions per guardar fragments de l'expressió reconstruïts. Durant la lectura caldrà tenir en compte la prioritat i l'associativitat dels operadors.

Llegirem cada element de l'expressió infixa i:

- Si és un (, l'apilem a la pila d'operadors.
- Si és un), desapilarem els operadors de la pila d'operadors fins que trobem un

[1]https://en.wikipedia.org/wiki/Shunting_yard_algorithm

(: Per a cada operador crearem una expressió amb l'operador com arrel i una o dues expressions de la pila d'expressions com a fills. Les expressions usades les desapilarem. A continuació apilarem l'expressió creada a la pila d'expressions.

- Si és un operand, crearem una expressió que només conté l'operand i l'apilarem a la pila d'expressions.
- Si és un operador:
  - Mentre l'operador té prioritat inferior (o igual i té l'associativitat d'esquerra a dreta) que el capdamunt de la pila d'operadors, desapilarem l'operador de la pila i crearem una expressió amb l'operador com arrel i una o dues expressions de la pila d'expressions com a fills, les expressions usades les desapilarem i continuació apilarem l'expressió creada a la pila d'expressions.
  - Sempre al final hem d'apilar l'operador d'entrada a la pila d'operadors.

Quan s'acabi de llegir l'expressió infixa, cal desapilar tots els operadors que quedin: Per a cada operador, crearem una expressió amb l'operador com arrel i una o dues expressions de la pila d'expressions com a fills. Les expressions usades les desapilarem. A continuació apilarem l'expressió creada a la pila d'expressions.

Finalment dins de la pila d'expressions tindrem un únic element amb l'expressió complerta reconstruïda (si l'expressió infixa d'entrada era correcta).

2.5 Avaluació i regles

L'avaluació d'una expressió es calcula d'acord amb una sèrie de regles. Les regles s'apliquen sistemàticament, fins que no pot simplificar-se més l'expressió. Convé distingir entre *valor* i *avaluació*, en particular quan parlem de variables, atès que no tenen perquè coincidir. Considerem la següent seqüència d'instruccions:

```
> x:= a * b
a*b
> a:= 3
3
> x
3*b
> b:= z+1
z+1
> x
3*(z+1)
> unassign a
a
```

```
> unassign b
b
> x
a*b
> 2.0*e
2.0*e
> evalf(% + 1)
6.4365637
```

La primera assignació li dóna a x el valor $a \cdot b$. Aquest és el seu valor durant tota la sessió, ja que no tornarem a canviar el valor. Però l'avaluació de x varia durant la sessió. Després d'assignar-li el valor, la seva avaluació seria $a \cdot b$. Després de fer $a := 3$ l'avaluació de x és $3b$, i després de canviar el valor de b tenim que l'avaluació de x és $3(z + 1)$. I per últim, un cop desassignades les variables a i b l'avaluació de x torna a coincidir amb el seu valor inicial, això és, $a \cdot b$.

Les dues últimes comandes mostren que la constant predefinida `e` es comporta com una variable sense valor assignat; però `evalf(e)` sí avalua a la corresponent constant real (veure la regla 10 sobre l'avaluació d'`evalf`).

Per constants i variables les regles per determinar el seu valor o avaluació són:

1. El valor i l'avaluació d'una expressió constant numèrica és la pròpia constant (en el cas de la constant predefinida `e` el seu valor i la seva avaluació és el seu nom).
2. El valor d'una variable és l'avaluació de l'última expressió que se li hagi assignat; en el seu defecte, el seu nom. L'avaluació d'una variable és l'avaluació del seu valor en aquest instant.
3. En la comanda $x := E$ s'imprimeix el resultat de l'avaluació d'E; addicionalment s'assigna l'avaluació de E com a valor de la variable x.

En les regles següents c, c', c_1, c_2, ... denotan constants numèriques (exclosa la constant predefinida `e`), x, y i z s'usaran per denotar variables i E, E', E_1, ... per denotar expressions arbitràries.

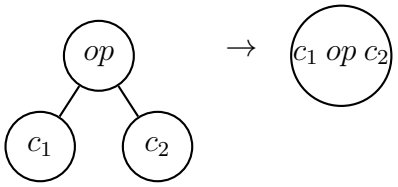
Usem equacions del tipus $E \to E'$ per indicar que l'expressió E es simplifica a l'expressió E'; ocasionalment, si resulta més clar, usarem un diagrama per expressar la regla.

A més totes les constants que apareguin a les regles de manera explícita (p.e. 0, 1 ó 2) són enteres, llevat que es digui el contrari. Així per exemple $\texttt{log}(1.0)$ no pot simplificar-se amb la regla $9e$ (encara que sí amb la regla $9k$), i no podem simplificar $3 - 0.0$ (ja que no considerem que $0 = 0.0$), ni tampoc $3 - 3.0$. En canvi, $\texttt{log}(4/4)$ simplifica a 0, ja que $4/4$ es simplifica amb la regla número 1 al racional $1/1$, després amb la regla 0 se simplifica a l'enter 1 i finalmente es podrà aplicar la regla que diu que $\texttt{log}(1) = 0$.

Regles:

0. Tota constant racional $c = n/d$ el denominador d de la qual val 1 (p.e. -2/1) es simplifica a la corresponent constant entera $c' = n$.

1. Per qualsevol operació binària $op \in \{+, -, *, /\}$ i qualssevol constants c_1 i c_2 del mateix tipus,



   En el cas que les constants c_1 i c_2 siguin enters i l'operació op sigui la divisió llavors $c_1\ op\ c_2$ és una constant racional si el quocient no és exacte. Si una de les constants és entera i l'altra racional la regla també pot ser aplicada convertint prèviament la constant entera al seu equivalent racional. Es produeix un error si $op = /$ i $c_2 = 0$ (com enter, racional o real).

2. Regles de suma:
   (a) $0 + E \to E$
   (b) $E + 0 \to E$
   (c) $E + E \to 2 * E$
   (d) $E + (-E') \to E - E'$
   (e) $(-E') + E \to E - E'$
   (f) $E_1 * E + E_2 * E \to (E_1 + E_2) * E$
   (g) $E_1 * E + E * E_2 \to (E_1 + E_2) * E$
   (h) $E * E_1 + E_2 * E \to (E_1 + E_2) * E$
   (i) $E * E_1 + E * E_2 \to (E_1 + E_2) * E$
   (j) $E_1/E + E_2/E \to (E_1 + E_2)/E$

3. Regles de resta:
   (a) $0 - E \to -E$ (canvi de signe)
   (b) $E - 0 \to E$
   (c) $E - E \to 0$
   (d) $E - (-E') \to E + E'$
   (e) $E_1 * E - E_2 * E \to (E_1 - E_2) * E$
   (f) $E_1 * E - E * E_2 \to (E_1 - E_2) * E$
   (g) $E * E_1 - E_2 * E \to (E_1 - E_2) * E$

(h) $E * E_1 - E * E_2 \rightarrow (E_1 - E_2) * E$

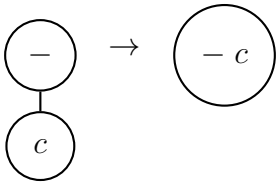
(i) $E_1/E - E_2/E \rightarrow (E_1 - E_2)/E$

4. Regles del canvi de signe i del signe positiu:

   (a) $-(-E) \rightarrow E$

   (b) $+E \rightarrow E$

   (c) Per qualsevol constant c (entera, racional o real)



5. Regles de multiplicació:

   (a) $1 * E \rightarrow E$

   (b) $E * 1 \rightarrow E$

   (c) $0 * E \rightarrow 0$

   (d) $E * 0 \rightarrow 0$

   (e) $E * E \rightarrow E\hat{}2$

   (f) $E * (-E') \rightarrow -(E * E')$

   (g) $(-E) * E' \rightarrow -(E * E')$

   (h) $E * (1/E') \rightarrow E/E'$

   (i) $(1/E') * E \rightarrow E/E'$

   (j) $E_1\hat{}E * E_2\hat{}E \rightarrow (E_1 * E_2)\hat{}E$

   (k) $E\hat{}E_1 * E\hat{}E_2 \rightarrow E\hat{}(E_1 + E_2)$

   (l) $\texttt{exp}(E) * \texttt{exp}(E') \rightarrow \texttt{exp}(E + E')$

6. Regles de divisió:

   (a) $E/0$ és un error.

   (b) $0/E \rightarrow 0$

   (c) $E/1 \rightarrow E$

   (d) $E/E \rightarrow 1$

   (e) $E/(-E') \rightarrow -(E/E')$

   (f) $E/(1/E') \rightarrow E * E'$

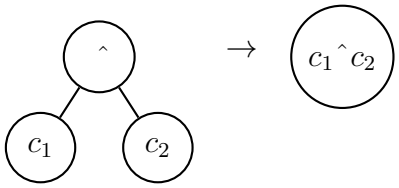
   (g) $E_1\hat{}E/E_2\hat{}E \rightarrow (E_1/E_2)\hat{}E$

   (h) $E\hat{}E_1/E\hat{}E_2 \rightarrow E\hat{}(E_1 - E_2)$

   (i) $\texttt{exp}(E)/\texttt{exp}(E') \rightarrow \texttt{exp}(E - E')$

7. Regles de exponenciació:

   (a) Si $c2$ és una constant de tipus enter, o si $c1$ i $c2$ son constants reals llavors

   Es produeix un error si $c1$ és una constant negativa (entera, racional o real) i $c2$ no és una constant entera.

   (b) $E\hat{}0 \rightarrow 1$

   (c) $E\hat{}1 \rightarrow E$

   (d) $E\hat{} - E' \rightarrow 1/(E\hat{}E')$

   (e) $(E_1\hat{}E_2)\hat{}E_3 \rightarrow E_1\hat{}(E_2 * E_3)$

   (f) $(\texttt{exp}(E))\hat{}E' \rightarrow \texttt{exp}(E * E')$

8. Regles de radicació (funció `sqrt`):

   (a) $\texttt{sqrt}(c) \rightarrow \texttt{error}$, per qualsevol constant entera, racional o real $c < 0$

   (b) $\texttt{sqrt}(0) \rightarrow 0$

   (c) Si la funció `sqrt` està aplicada sobre una constant en coma flotant es calcula d'immediat el seu resultat usant la funció corresponent de la biblioteca matemàtica de C++, i en concret es pendrà el valor positiu (p.e. tant 3.0 com -3.0 són valors correctes per $\texttt{sqrt}(9.0)$, però ens quedarem amb el primer).

   (d) $\texttt{sqrt}(E) \rightarrow E\hat{}\frac{1}{2}$ (aquí, una meitat es representa com constant racional, no com una expressió amb tres tokens)

9. Regles de les funcions exponencial i logaritme (natural):

   (a) $\texttt{exp}(0) \rightarrow 1$

   (b) $\texttt{exp}(1) \rightarrow \texttt{e}$

   (c) $\texttt{exp}(\texttt{log}(E)) \rightarrow E$

   (d) $\texttt{log}(c) \rightarrow \texttt{error}$, per qualsevol constant entera, racional o real $c \leq 0$

   (e) $\texttt{log}(1) \rightarrow 0$

   (f) $\texttt{log}(\texttt{e}) \rightarrow 1$

   (g) $\texttt{log}(\texttt{exp}(E)) \rightarrow E$

   (h) $\texttt{log}(E * E')) \rightarrow \texttt{log}(E) + \texttt{log}(E')$

   (i) $\texttt{log}(E/E')) \rightarrow \texttt{log}(E) - \texttt{log}(E')$

   (j) $\texttt{log}(E\hat{}E')) \rightarrow E' * \texttt{log}(E)$

   (k) Si la funció `exp` o `log` estan aplicades sobre una constant en coma flotant es calcula d'immediato el seu resultat usant les funcions corresponents de la biblioteca matemàtica de C++.

10. L'operació `evalf` retorna quelcom diferent depenent al que s'estigui aplicant:

    - Si s'aplica sobre una constant en coma flotant torna la pròpia constant.
    - Si s'aplica sobre un enter realitza la conversió, i si s'aplica sobre un racional es calcula el quocient real entre el numerador i el denominador.
    - Si s'aplica sobre la constant `e` retorna el valor 2.718281828.
    - Si s'aplica sobre una variable x que no té valor assignat retorna la pròpia variable.
    - Per la resta, la seva aplicació és recursiva: $\texttt{evalf}(E\ op\ E') \to \texttt{evalf}(E)\ op\ \texttt{evalf}(E')$ si op és una operació binària i $\texttt{evalf}(f(E)) \to f(\texttt{evalf}(E))$ per una funció f qualsevol exceptuant la pròpia `evalf`.

2.6 Procés d'avaluació

La tasca més complexa que realitza ESINMATH és l'avaluació d'expressions. Durant una sessió els valors assignats a les variables s'utilitzen en l'avaluació d'expressions de les que elles en formen part. Simplificant, podem definir l'"estat" d'una sessió ESINMATH com els valors assignats a les variables en cada moment.

Però en el procés d'avaluació no intervenen només els valors assignats a les variables: s'ha definit un conjunt de regles de simplificació que permeten manipular expressions i obtenir d'altres més "senzilles". El propòsit del procés d'avaluació consisteix doncs en obtenir una expressió final on, per una banda totes les variables que apareguin en ella no tinguin valor assignat, i per una altra, l'expressió estigui completament simplificada. Una expressió que compleix aquests dos requeriments diem que s'ha reduït a una "forma normal".

Una propietat desitjable d'ESINMATH seria que l'avaluació d'una expressió doni una única expressió, sigui quin sigui l'ordre dels passos seguits durant el procés d'avaluació, o, el que és el mateix, voldríem que la "forma normal" calculada fos única. Desafortunadament no sempre és així; veiem dos exemples en els que això no passa. En ells es mostren en vermell les variables que són sustituïdes pels seus valors i també les subexpressions a les quals s'aplica una regla de simplificació:

1. $E = x - - 3$
   a) $x - - [3] \Rightarrow x - [-3]$ apliquem la regla $-[c] \to [-c]$
   b) $x - - [3] \Rightarrow x + [3]$ apliquem la regla $E - - E' \to E + E'$

2. $E = 0 * (1/z)$ on $z = 0$
   a) $0 * (1/z) \Rightarrow 0$ apliquem la regla $0 * E \to 0$
   b) $0 * (1/z) \Rightarrow 0 * (1/0) \Rightarrow$ error subtituïm la variable z y apliquem la regla $E/0 \to \texttt{error}$

Per assegurar que el procés d'avaluació d'una expressió E acaba en una expressió única us indiquem els passos i l'ordre que heu de seguir:

a) en primer lloc s'han de sustituir totes les aparicions de variables d'E per les expressions que tinguin assignades en aquest moment. Aquest procés s'ha d'iterar fins que totes les variables de l'expressió obtinguda E' siguin variables que no tinguin valor assignat. L'ordre en el que s'apliquen les substitucions no és rellevant. Es poden produir situacions de recursió infinita però **no és obligatori detectar-les**; cal tenir en compte que si no es fa aquesta detecció el procés de substitucions de les variables pot no acabar. Per exemple, si $E = a + c$ on $a = c$ i $c = a + 1$ l'aplicació de les substitucions produirà successives expressions de la forma $a + c$, $c + c$, $(a+1)+(a+1)$, $(c+1)+(c+1)$, $((a+1)+1) + ((a+1)+1)$, ... Cadenes infinites como aquesta, o similars, es produiran sigui quin sigui el criteri que utilitzem per aplicar les substitucions. La raó radica en que existeix una circularitat en les substitucions que involucren a les variables a i c.
El cas elemental de circularitat es dóna quan el valor assignat a una variable x és una expressió que conté a x. Perquè aquesta classe de circularitat no es produeixi no permetrem assignacions que generin aquesta situació, tal i com es mostra en el següent exemple:

```
> x := 3
3
> x := x + 1
4
> unassign x
x
> x := x + 1
Error: Assignació amb circularitat infinita.
```

b) a continuació aplicareu les regles de simplificació a $E_0 = E'$ obtenint a cada pas de simplificació una expressió E_{i+1} fins que no podem aplicar més regles de simplificació. L'expressió final obtinguda E_n serà el resultat de l'avaluació de l'expressió inicial E.
El criteri que usarem per simplificar cadascuna de les expressions E_i serà seguir un recorregut *left to right* postordre fins trobar una subexpressió d'E_i a la que es pugui aplicar alguna regla de simplificació.
Però encara hi ha una font d'indeterminisme: és possible poder aplicar més d'una regla de simplificació a la subexpressió de E_i seleccionada en aquest pas. Si es produeix un conflicte com aquest, i ateses que les regles de simplificación estan ordenades, aplicareu l'especificada en primer lloc.

A continuació veiem un exemple del procés d'avaluació de l'expressió $E = a + b * c$ on les variables tenen assignats els següents valors: $a = 0$, $b = c$ i $c = \texttt{sqrt}(d)$

a) Aplicació de les substitucions. Per exemple (aquí l'ordre no és rellevant):

$a + b * c$	substitueix c por $\texttt{sqrt}(d)$
$a + b * \texttt{sqrt}(d)$	substitueix a por 0
$0 + b * \texttt{sqrt}(d)$	substitueix b por c
$0 + c * \texttt{sqrt}(d)$	substitueix c por $\texttt{sqrt}(d)$
$0 + \texttt{sqrt}(d) * \texttt{sqrt}(d)$	no és possible fer més substitucions

Per tant, tenim que $E' = 0 + \texttt{sqrt}(d) * \texttt{sqrt}(d)$

b) Passos de simplificació. L'ordre és necessàriament el següent (al costat de cada pas s'indica la regla aplicada per passar a la següent expressió):

1.	$E_0 = 0 + \texttt{sqrt}(d) * \texttt{sqrt}(d)$	regla 8d: $\texttt{sqrt}(E) \to E\hat{}\frac{1}{2}$
2.	$E_1 = 0 + d\hat{}\frac{1}{2} * \texttt{sqrt}(d)$	regla 8d: $\texttt{sqrt}(E) \to E\hat{}\frac{1}{2}$
3.	$E_2 = 0 + d\hat{}\frac{1}{2} * d\hat{}\frac{1}{2}$	regla 5e: $E * E \to E\hat{}2$
4.	$E_3 = 0 + d\hat{}\frac{1}{2}\hat{}2$	regla 7e: $(E_1\hat{}E_2)\hat{}E_3 \to E_1\hat{}(E_2 * E_3)$
5.	$E_4 = 0 + d\hat{}(\frac{1}{2} * 2)$	regla 1: racional $*$ enter $\to$ racional
6.	$E_5 = 0 + d\hat{}\frac{1}{1}$	regla 0: racional $\to$ enter
7.	$E_6 = 0 + d\hat{}1$	regla 7c: $E\hat{}1 \to E$
8.	$E_7 = 0 + d$	regla 2a: $0 + E \to E$
9.	$E_8 = d$	forma normal

Finalment tenim que $E_8 = d$ és l'expressió resultant de l'avaluació de l'expressió inicial $E = a + b * c$.

3 Disseny modular

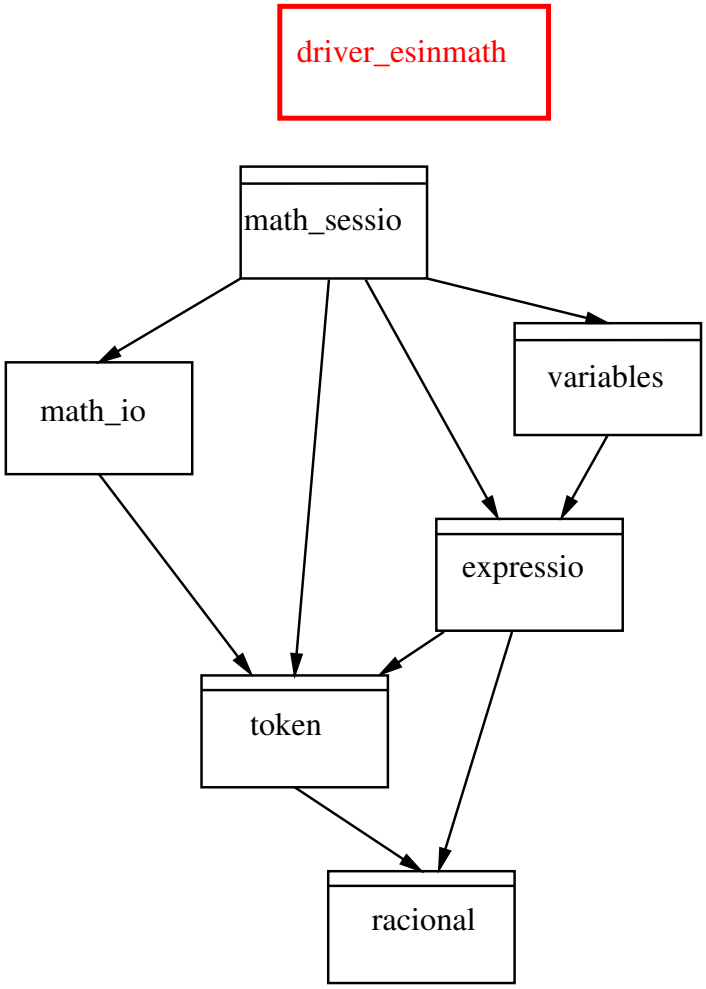


Figura 2: Disseny modular del projecte.

El projecte consisteix en la implementació del disseny modular que es descriu en aquesta secció. La vostra missió serà dissenyar i implementar les classes racional, token, expressio, variables i math_sessio, i el mòdul math_io.

S'han omès d'aquest diagrama (Figura 2) la classe error i el mòdul util de la biblioteca `libesin` per claretat, ja que moltes classes i mòduls del diagrama usen aquests mòduls. La classe error està documentada en el *Manual de laboratori d'ESIN*, i el mòdul util està documentat on-line en el fitxer `esin/util`. En algunes classes d'aquest projecte s'usa també les classes string i math de la biblioteca estàndar de C++. Aquestes relacions d'ús i les classes en qüestió tampoc es mostren a la figura. A tots els mòduls o classes d'aquesta pràctica es pot usar també la classe string de la llibreria estàndard de C++.

Recordeu que no es pot utilitzar cap classe d'una biblioteca externa en les vostres classes, exceptuant si en aquesta documentació s'indica el contrari.

També tindreu a la vostra disposició, més endavant, els jocs de proves públics i el mòdul driver_esinmath. Aquest mòdul conté el programa principal i s'us dóna implementat. Usa tots els altres mòduls i classes (permet invocar cadascuna de les operacions dels diferents mòduls i classes) encara que no s'han representat les fletxes d'ús explícitament.

A totes les classes cal implementar els mètodes de construcció per còpia, assignació i destrucció davant la possibilitat que empreu memòria dinàmica en la classe en qüestió. Si no s'usa memòria dinàmica, la implementació d'aquests tres mètodes és molt senzilla doncs n'hi haurà prou amb imitar el comportament del que serien els corresponents mètodes d'ofici (el destructor no fa "res", i els altres fan còpies atribut per atribut).

Així mateix us proporcionem tots els fitxers capçalera (`.hpp`) d'aquest disseny modular. No podeu crear els vostres propis fitxers capçalera ni modificar de cap manera els que us donem. Tingueu present que heu de respectar escrupolosament l'especificació de cada classe que apareix en el corresponent `.hpp`.

ATENCIÓ: És important que una cop implementat cadascuna de les classes, les sotmeteu als vostres jocs de proves. A més, també és fonamental dissenyar amb força detall la representació de cada clase i els seus algorismes sobre paper, i prendre notes de tots els passos seguits abans de començar a codificar. Aquesta informació serà vital de cara a la preparació de la documentació final.

En resum, en aquest projecte les tasques que heu de realitzar i l'ordre que heu de seguir és el següent:

- Implementar la classe racional
- Implementar la classe token
- Implementar la classe expressio
- Implementar la classe variables
- Implementar el mòdul math_io
- Implementar la classe math_sessio

4 La classe token

Un objecte de la classe `token` conté un operand, un operador o funció predefinida, o un símbol de "puntuació"(parèntesi o coma).

Els operands poden ser constants enteres, racionals o reals (en coma flotant), la constant predefinida `e`, variables o la variable predefinida (`%`).

Els operadors admissibles són la suma (+), resta (−), multiplicació (*), divisió (/), exponenciació (^), canvi de signe (−) i signe positiu (+).

Les funcions predefinides són l'arrel quadrada (`sqrt`), exponencial (`exp`), logaritme (`log`) i avaluació en coma flotant (`evalf`).

Hi ha un altre tipus de tokens, anomenats comandes, que són: assignació (`:=`), desassignació (`unassign`) i final de sessió (`byebye`).

Un objecte de la classe `token` conté informació del tipus de token(que pot ser qualsevol dels valors que apareixen llistats en l'enumeració). En el cas de tractar-se d'un token de tipus `CT_ENTERA`, `CT_RACIONAL` o `CT_REAL` també emmagatzema el seu valor enter, `racional` o real corresponent. Si el token és una VARIABLE guarda l'string amb el seu nom.

Implementació: La representació d'aquesta classe es trobarà en el fitxer `token.rep` i la implementació en el fitxer `token.cpp`. La documentació serà mínima, donada la senzillesa de les dades i els mètodes.

```
#ifndef _TOKEN_HPP
#define _TOKEN_HPP
#include <string>
#include <esin/error>
#include <esin/util>
#include "racional.hpp"

class token {
public:

  enum codi {NULLTOK,
             CT_ENTERA, CT_RACIONAL, CT_REAL, CT_E,
             VARIABLE, VAR_PERCENTATGE,
             SUMA, RESTA, MULTIPLICACIO, DIVISIO, EXPONENCIACIO,
             CANVI_DE_SIGNE, SIGNE_POSITIU,
             SQRT, EXP, LOG, EVALF,
             OBRIR_PAR, TANCAR_PAR, COMA,
             ASSIGNACIO, DESASSIGNACIO, BYEBYE};
```

```
/*Constructores: Construeixen tokens pels operadors, les constants enteres,
  les constants racionals, les constants reals i les variables(el seu
  identificador), respectivament.

  La primera constructora s'utiliza para aquells tokens que són operadors,
  funcions predefinides, noms de comandes o símbols de ''puntuació''; per
  tant es produeix un error si el codi és CT_ENTERA, CT_RACIONAL, CT_REAL o
  VARIABLE.

  L'última constructora torna un error si l'string donat no és un
  identificador vàlid per una variable: ha d'estar format exclusivament per
  lletres majúscules, minúscules o caràcters de subratllat _. Dit d'una altra
  forma, només conté caràcters els codis ASCII dels quals estan entre 65
  ('A') i 90('Z'), entre 97 ('a') i 122 ('z') o el 95 ('_') i no pot
  coincidir amb un nom reservat: unassign, e, sqrt, log, exp i evalf.*/
  explicit token(codi cod = NULLTOK) throw(error);
  explicit token(int n) throw(error);
  explicit token(const racional & r) throw(error);
  explicit token(double x) throw(error);
  explicit token(const string & var_name) throw(error);

  // Constructora por còpia, assignació i destructora.
  token(const token & t) throw(error);
  token & operator=(const token & t) throw(error);
  ~token() throw();

/*Consultores: Retornen respectivament el codi i el valor (en el cas de
  constants enteres, racionals o reals) o l'identificador (en el cas de
  variables). Es produeix un error si apliquem una consultora inadequada
  sobre un token, p.e. si apliquem valor_enter sobre un token que no sigui
  una CT_ENTERA.*/
  codi tipus() const throw();
  int valor_enter() const throw(error);
  racional valor_racional() const throw(error);
  double valor_real() const throw(error);
  string identificador_variable() const throw(error);

/*Igualtat i desigualtat entre tokens. Dos tokens es consideren iguals si els
  seus codis ho són i si 1) en cas de ser CT_ENTERA, CT_RACIONAL o CT_REAL,
  els seus valors són iguals i 2) en cas de ser VARIABLE, tenen el mateix
  nom. */
  bool operator==(const token & t) const throw();
  bool operator!=(const token & t) const throw();

/*Precedència entre tokens. L'operador > retorna cert si i només si el token
  és un operador amb major precedència que l'operador del token t. Si algun
  dels tokens no és un operador es produeix un error.*/
  bool operator>(const token & t) const throw(error);
```

```
  bool operator<(const token & t) const throw(error);

  // Gestió d'errors.
  static const int IdentificadorIncorrecte      = 11;
  static const int ConstructoraInadequada       = 12;
  static const int ConsultoraInadequada         = 13;
  static const int PrecedenciaEntreNoOperadors  = 14;

private:
  #include "token.rep"
};
#endif
```

5 La classe racional

La classe `racional` emmagatzema un nombre racional.

Decisions sobre les dades: Un objecte de la classe `racional` sempre estarà en la seva versió simplificada, és a dir, on el numerador i el denominador són primers entre sí i el signe del racional està en el numerador (el denominador és sempre positiu). Si el numerador és 0 el denominador de la versió simplificada és necessàriament 1. Per exemple, la versió simplificada del racional (10/-5) és (-2/1).

Implementació: La representació d'aquesta classe es trobarà en el fitxer `racional.rep` i la implementació en el fitxer `racional.cpp`. La documentació serà mínima, donada la senzillesa de les dades i els mètodes.

```
#ifndef _RACIONAL_HPP
#define _RACIONAL_HPP
#include <esin/error>
#include <esin/util>

using std::string;

class racional {
public:

  // Constructora. Construeix un racional en la seva versió simplificada.
  // Es produeix un error si el denominador és 0.
  explicit racional(int n=0, int d=1) throw(error);

  // Constructora per còpia, assignació i destructora.
  racional(const racional & r) throw(error);
  racional & operator=(const racional & r) throw(error);
  ~racional() throw();

  // Consultores. La part_entera d'un racional pot ser
  // positiva o negativa. El residu SEMPRE és un racional positiu.
  int num() const throw();
  int denom() const throw();
  int part_entera() const throw();
  racional residu() const throw();

  /* Sobrecàrrega d'operadors aritmètics. Retorna un racional en la seva
     versió simplificada amb el resultat de l'operació. Es produeix un
     error al dividir dos racionals si el segon és 0.*/
  racional operator+(const racional & r) const throw(error);
  racional operator-(const racional & r) const throw(error);
  racional operator*(const racional & r) const throw(error);
  racional operator/(const racional & r) const throw(error);
```

```
  /* Sobrecàrrega de operadors de comparació. Retornen cert, si i només si
     el racional sobre el que s'aplica el mètode és igual (==), diferent
     (!=), menor (<), menor o igual (<=), major (>) o major o igual (>=)
     que el racional r.*/
  bool operator==(const racional & r) const throw();
  bool operator!=(const racional & r) const throw();
  bool operator<(const racional & r) const throw();
  bool operator<=(const racional & r) const throw();
  bool operator>(const racional & r) const throw();
  bool operator>=(const racional & r) const throw();

  // Gestió d'errors.
  static const int DenominadorZero = 21;

private:
  #include "racional.rep"
};
#endif
```

6 La classe expressio

Els objectes de la classe `expressio` permeten representar i manipular expressions inicialment creades a partir d'una seqüència de tokens.

Decisions sobre les dades: El nombre de tokens a partir del qual es construeix una expressió no està acotat.

Implementació: La representació d'aquesta classe es trobarà en el fitxer `expressio.rep` i la implementació en el fitxer `expressio.cpp`. En la implementació es poden usar les classes `list` i `stack` de l'*STL*, en cas necessari. En la documentació d'aquesta classe no serà necessari que indiqueu el cost dels mètodes.

És recomenable que en primer lloc desenvolupeu gradualment la implementació d'aquesta classe. Comenceu amb l'aplicació de les regles de la suma, després les del producte, etc. La implementació arribarà un moment que hauria de processar adequadament les expressions que involucrin exclusivament constants enteres i racionals, variables i els operadors '+', '-' (canvi de signe), '-' (resta), '*' i '/'. Un cop aconseguit això, incorporeu el codi necessari per processar la resta d'operadors i funcions, les constants reals, ...

```
#ifndef _EXPRESSIO_HPP
#define _EXPRESSIO_HPP
#include <list>
#include <string>
#include <esin/error>
#include <esin/util>
#include "token.hpp"
#include "racional.hpp"

using std::list;
using std::string;

class expressio {
public:

  /* Constructora d'una expressió formada per un sol token: un operand. Si
     s'utiliza el valor del token per defecte es construeix la que
     anomenem "expressió buida". Si el tipus del token no és el del token
     per defecte (NULLTOK), ni el d'una CT_ENTERA, CT_RACIONAL, CT_REAL,
     CT_E, VARIABLE o VAR_PERCENTAtGE es produeix un error sintàctic. */
  expressio(const token t = token()) throw(error);

  /* Constructora a partir d'una seqüència de tokens. Es produeix un error si
     la seqüència és buida o si no es pot construir l'arbre d'expressió
     corresponent(és a dir, si és sintàcticament incorrecta). */
  expressio(const list<token> & l) throw(error);
```

```
// Constructora per còpia, assignació i destructora.
expressio(const expressio & e) throw(error);
expressio & operator=(const expressio & e) throw(error);
~expressio() throw(error);

// Retorna cert si i només si s'aplica a l'expressió buida.
operator bool() const throw();

/* Operadors d'igualtat i desigualtat. Dues expressions es consideren
   iguals si i només si els seus arbres d'expressió són idèntics. */
bool operator==(const expressio & e) const throw();
bool operator!=(const expressio & e) const throw();

/* Retorna una llista sense repeticions, en qualsevol ordre, amb
   els noms de les variables de l'expressió. */
void vars(list<string> & l) const throw(error);

/* Substitueix totes les aparicions de la variable de nom v per
   l'expressió e. Si no existeix la variable v dins de l'expressió a la
   que apliquem aquest mètode l'expressió no es modifica. */
void apply_substitution(const string & v, const expressio & e) throw(error);

/* Aplica un pas de simplificació a l'expressió. La subexpressió a
   simplificar es busca seguint el recorregut "left to right" postordre
   explicat a l'apartat "Procés d'avaluació". Es pot produir qualsevol dels
   errors semàntics que apareixen més avall numerats des del 32 al 35. */
void simplify_one_step() throw(error);

/* Aplica successius passos de simplificació com l'anterior fins que
   l'expressió es trobi completament simplificada. Llavors diem que es
   troba en "forma normal". Es pot produir qualsevol dels errors
   semàntics que apareixen més avall numerats des del 32 al 35. */
void simplify() throw(error);

/* Converteix l'expressió en la seqüència de tokens lt corresponent: els
   operadors apareixen entre els operands si són infixos, abans si són
   prefixos i els símbols de funció van seguits de parèntesis que tanquen
   els seus arguments separats per comes. S'han d'introduir només aquells
   parèntesis que siguin estrictament necessaris per trencar les regles de
   precedència o associativitat en l'ordre d'aplicació dels operadors. */
void list_of_tokens(list<token> & lt) throw(error);

// Gestió d'errors.
static const int ErrorSintactic   = 31;
static const int NegatElevNoEnter = 32;
static const int DivPerZero       = 33;
static const int LogDeNoPositiu   = 34;
static const int SqrtDeNegatiu    = 35;
```

```
private:
  #include "expressio.rep"
};
#endif
```

7 La classe variables

La classe `variables` és un diccionari de parells $< clau,\ valor >$ on la clau és un `string` que representa el nom d'una variable i el valor és l'expressió que té assignada.

Decisions sobre les dades: Un objecte de la classe `variables` conté un nombre no acotat de variables, on totes són diferents i sempre tenen un valor associat.

Implementació: La representació d'aquesta classe es trobarà en el fitxer `variables.rep` i la implementació en el fitxer `variables.cpp`. La classe `list` de l'*STL* es pot usar únicament per retornar les dades del mètode `dump` i per tant no pot ser usada amb un altre objectiu.

Els mètodes d'aquesta classe han de ser tots eficients en el cas pitjor o mig. En particular, el constructor per còpia, assignació i destructor han de tenir cost $O(N)$, on N és el nombre de variables. La resta, exceptuant `dump`, ha de tenir cost $O(\log(N))$ o inferior. El mètode `dump` es fa molt ocasionalment.

```
#ifndef _VARIABLES_HPP
#define _VARIABLES_HPP
#include <list>
#include <string>
#include <esin/error>
#include <esin/util>
#include "expressio.hpp"

using std::list;
using std::string;

class variables {
public:

  // Construeix un conjunt de variables buit.
  variables() throw(error);

  // Constructora per còpia, assignació i destructora.
  variables(const variables & v) throw(error);
  variables & operator=(const variables & v) throw(error);
  ~variables() throw(error);

  /* Afegeix al conjunt de variables la variable de nom v juntament amb el seu
     valor-expressió e. Si la variable v ja existia llavors li assigna el nou
     valor-expressió. */
  void assign(const string & v, const expressio & e) throw(error);

  /* Elimina del conjunt de variables el parell amb la variable de nom v. Si
     la variable no existeix llavors no fa res. */
  void unassign(const string & v) throw();
```

```
  /* Consulta el valor-expressió de la variable v. Si la variable no està en
     el conjunt de variables retorna l'expressió buida. */
  expressio valor(const string & lv) const throw(error);

  /* Retorna en l totes les claus del conjunt de variables, en un ordre
     qualsevol. Si no hi ha cap clau retorna la llista buida.*/
  void dump(list<string> & l) const throw(error);

private:
  #include "variables.rep"
};
#endif
```

8 La classe math_sessio

La classe `math_sessio` conté la informació necessària per desenvolupar una sessió del programa ESINMATH. Guarda en tot moment l'estat de la sessió i executa comandes que consulten i/o modifiquen aquest estat.

Implementació: La representació d'aquesta classe estarà en el fitxer `math_sessio.rep` i la implementació en el fitxer `math_sessio.cpp`. L'ús de la classe `list` de l'*STL* és exclusivament per l'entrada i sortida de dades de diversos mètodes de la classe i no pot ser usada amb una altra finalitat.

A l'igual que a la classe expressio, no caldrar indicar el cost dels mètodes.

```
#ifndef _MATH_SESSIO_HPP
#define _MATH_SESSIO_HPP
#include <list>
#include <esin/error>
#include <esin/util>
#include "token.hpp"
#include "expressio.hpp"
#include "variables.hpp"
#include "math_io.hpp"

using std::list;

class math_sessio {
public:

  /* Constructora. Crea una nova sessió buida i emmagatzema a la variable
     especial % l'expressió buida. */
  math_sessio() throw(error);

  // Constructora per còpia, assignació i destructora.
  math_sessio(const math_sessio & es) throw(error);
  math_sessio & operator=(const math_sessio & es) throw(error);
  ~math_sessio() throw(error);

  /* Aquest mètode rep una llista de tokens, lin, lèxicament correcta.
     Primerament analitza parcialment lin per verificar si la comanda és
     correcta. Si és correcta executa la comanda que conté lin.
     Les comandes són:
      * avaluació d'una expressió E.
      * assignació v :=  E. S'avalua E i s'assigna el resultat a la variable
        de nom v.
      * desassignació d'una variable v: unassign v.
      * final de sessió: byebye

     En l'anàlisi de la comanda lin es produeix un error de comanda
```

```
      incorrecta en els següents casos:
       * si conté el token DESASSIGNACIO i,
         * o bé la comanda no té dos tokens
         * o bé aquest no és el primer token
         * o bé el segon token no és una VARIABLE.
       * si conté el token BYEBYE i aquest no és el primer i únic token
         de la comanda.
       * si conté el token ASSIGNACIO i,
         * o bé la comanda té longitud menor que dos
         * o bé no és el segon token
         * o bé el primer token no és un token VARIABLE.

      Les comandes que involucren avaluació (avaluar i assignar) retornen
      l'expressió avaluada en forma de llista de tokens en lout. La comanda
      unassign retorna la llista que conté com únic token la variable
      desassignada. Finalment la comanda byebye retorna la llista buida.
      Es produeix un error en una assignació, si després l'avaluació de
      l'expressió es comprova que la variable assignada forma part del conjunt
      de variables de l'expressió avaluada, tal i com s'explica a
      l'apartat "Procés d'avaluació". */
    void execute(const list<token> & lin, list<token> & lout) throw(error);

    // Retorna cert si i només si la sessió ha finalitzat.
    bool end_of_session() const throw();

    /* Retorna en forma de llista d'strings, en un ordre qualsevol, la llista de
       variables juntament amb el seu valor assignat. Cada string de la llista
       té el format id = e, on id és el nom d'una variable i e és l'expressió
       (com string) assignada a id. Convertim una expressió e en el seu string
       corresponent invocant al mètode tostring del mòdul math_io. */
    void dump(list<string> & l) const throw(error);

    /* Donada una expressió e, aplica a les seves variables totes les
       substitucions definides per elles. L'expressió resultant només contindrà
       variables per les quals no hi ha substitució definida (no estan en el
       conjunt). Aquest procés s'explica en detall a l'apartat "Procés
       d'avaluació". S'assumeix que no existeix circularitat infinita entre les
       substitucions de les variables que formen part de l'expressió e. */
    void apply_all_substitutions(expressio & e) const throw(error);

    // Gestió d'errors.
    static const int SintComandaIncorrecta  = 51;
    static const int AssigAmbCirculInfinita = 52;

  private:
    #include "math_sessio.rep"
  };
  #endif
```

9 El mòdul math_io

El mòdul `math_io` conté un mètode `scan` que realitza l'anàlisi i descomposició d'un string en seqüència de tokens, i un altre `tostring` que realitza la transformació inversa.

Implementació: Aquest mòdul ja el teniu implementat en el fitxer `math_io.cpp`.

```
#ifndef _MATH_IO_HPP
#define _MATH_IO_HPP
#include <list>
#include <string>
#include <esin/error>
#include <esin/util>
#include "token.hpp"

using std::list;
using std::string;

namespace math_io {

  /* Realitza l'anàlisi lèxic de la comanda en forma d'string s, la converteix
     en la seqüència de tokens i retorna aquesta seqüència. Es produeix un
     error si existeix algun error lèxic, és a dir, algun caràcter que
     impedeixi el reconeixement d'un nou token. */
  void scan(const string & s, list<token> & lt) throw(error);

  // Converteix en un string la seqüència de tokens lt.
  string tostring(const list<token> & lt);

  // Gestió d'errors.
  const int ErrorLexic = 61;
};
#endif
```

10 Errors

Aquest fitxer conté els missatges d'error usats en la gestió d'errors.

```
11 token El nom de la variable no es valida.
12 token La constructora per aquest token no es l'adequada.
13 token Aquesta consultora del token no es apropiada.
14 token La precedencia es defineix entre operadors.

21 racional El denominador es zero.

31 expressio Error sintactic.
32 expressio Negatiu elevat a no enter.
33 expressio Divisio per zero.
34 expressio Logaritme de no positiu.
35 expressio Arrel quadrada de negatiu.

51 math_sessio La sintaxi de la comanda es incorrecta.
52 math_sessio Assignacio amb circularitat infinita.

61 math_io No es troba un nou token valid.
```

11 Documentació

Els fitxers lliurats han d'estar degudament documentats. És molt important descriure amb detall i precisió la representació escollida en el fitxer `.rep`, justificant les eleccions fetes, així com les operacions de cada classe. És especialment important explicar amb detall les representacions i els motius de la seva elecció comparant-la a possibles alternatives, i els algorismes utilitzats.

El cost en temps i en espai és freqüentment el criteri determinant en l'elecció, per la qual cosa s'hauran de detallar aquests costs en la justificació (sempre que això sigui possible) per a cada alternativa considerada i per a l'opció finalment escollida. A més caldrà detallar en el fitxer `.cpp` el cost de cada mètode públic i privat.

En definitiva heu de:

- comentar adequadament el codi, evitant comentaris inútils i superflus
- indicar, en la mesura que sigui possible, el cost dels mètodes de les classes (tant públics com privats)
- descriure amb detall i precisió la representació escollida i justifiqueu l'elecció respecte d'altres.

Un cop enviats els fitxers per via electrònica, aquests seran impresos per a la seva avaluació. No haureu d'imprimir-los vosaltres. No haureu de lliurar cap altra documentació. Per tal d'unificar l'aspecte visual del codi fem servir una eina de *prettyprinting* anomenada `astyle`. Podeu comprovar els resultats que produeix el *prettyprinter* mitjançant la comanda

```
% astyle --style=kr -s2 < fitxer.cpp > fitxer.formatejat
```

i a continuació podeu convertir-lo en un PDF per visualitzar-lo o imprimir-lo

```
% a2ps fitxer.formatejat -o - | ps2pdf - fitxer.pdf
```