UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Politècnica Superior d'Enginyeria
de Vilanova i la Geltrú

Programació 1 (PRO1)

Enunciat de la pràctica

CALCULADORA AMB MEMÒRIES

Autor: Jordi Esteve

Curs 2019/20 - Q2

Versió 1.0 - 2020.04.15

NORMATIVA DE LA PRÀCTICA

Aquesta és la normativa de la pràctica. És important que la llegiu i la seguiu. Incomplir la normativa pot suposar el suspens de l'assignatura.

1. La qualificació obtinguda a pràctiques fa mitjana ponderada amb la nota de teoria amb un pes del 35% sobre la nota total de l'assignatura.

2. La pràctica és obligatòria. La no presentació de la pràctica resultarà en un NO PRESENTAT com a nota final de l'assignatura.

3. La nota de la pràctica està formada per dues parts: programa i prova de validació individual (PVI). La nota de la PVI serà un número real entre 0 i 1. La fórmula per calcular la nota final de laboratori és la següent: NotaPrograma * NotaPVI.

4. La pràctica es realitzarà en equips de màxim dues persones. Serà motiu de no acceptació de la pràctica l'haver estat realitzada per més de dues persones.

5. Si dos o més treballs presentats són iguals, no serà acceptat **cap** dels treballs i implicarà un ZERO en la nota de l'assignatura. El criteri del professorat decidirà si dues pràctiques es poden considerar iguals o no. No s'admetran reclamacions sobre aquesta consideració.

6. Les pràctiques acceptades s'avaluaran mitjançant:

   - La correctesa i claredat del programa.
   - La correctesa de la documentació annexa i/o comentaris del programa.
   - L'execució del programa.

   Una pràctica que no funcioni pot ser igualment acceptada i avaluada pel professor.

7. El lliurament de la pràctica s'efectuarà a través del campus digital, en les condicions i format especificats a l'enunciat. L'incompliment de qualsevol d'aquestes condicions pot suposar la nota de NO PRESENTAT a la pràctica.

8. Un cop presentada la pràctica tindrà lloc la Prova de Validació Individual. La PVI consistirà en un control no presencial on haureu de programar una operació que serà una variació de les que heu fet a la pràctica. La PVI també pot tenir en compte els exercicis preparatoris realitzats a la plataforma jutge.org.

9. Els estudiants del mateix equip poden obtenir notes diferents a la PVI, resultant en notes de laboratori diferents per a cada un.

10. La no compareixença a la PVI d'un estudiant farà que la seva nota de laboratori sigui NO PRESENTAT, encara que hagi fet el lliurament de la pràctica.

11. Les consultes de la pràctica es faran al professor Jordi Esteve (jesteve@cs.upc.edu). Si són genèriques millor escriure-les al fòrum, així poden ser d'utilitat a altres companys.

1 Introducció

La pràctica consisteix en programar una calculadora d'expressions aritmètiques i booleanes que disposa de funcions avançades com un conjunt de memòries on guardar expressions i avaluar-les individualment o en cascada.

Les expressions poden contenir com operands números enters (tant positius com negatius), constants booleanes (`T`, `F`) i variables; i com operadors `not`, `and`, `or`, `==`, `!=`, `+`, `-`, `*`, `/`, `%`, `**`. Les variables estaran formades només per lletres minúscules; comencen sempre amb una lletra; després poden tenir lletres, dígits o el caràcter `_` (`x1`, `suma_items`, `y2_aux`).

Per exemple, algunes expressions que podrà gestionar la nostra calculadora són:

- not x + 4 != y - 8
- b != a or not n + 13 == -5 and p == T
- x * 5 / (14 + 21)

L'objectiu de la calculadora és llegir una sèrie d'expressions i avaluar-les o desar-les en memòries per ser avaluades posteriorment. L'avaluació pot donar com a resultat un valor enter o booleà, o simplement una expressió més senzilla que l'original, si és possible trobar-la. Per exemple, algunes avaluacions són:

Expressió	**Avaluació**
3 + (6 * 2)	15
x + 0 == 1 * x	T
6 * 2 + x != y - 32	12 + x != y - 32
a and T or not (b or F)	a or not b
x + y * 2 / (y - 27)	x + y * 2 / (y - 27)

1.1 Formats de representació d'expressions

Les expressions es poden escriure en tres notacions: infixa, prefixa, i postfixa.

La **notació infixa** és la més habitual, i és aquella en la que l'operador va enmig dels dos operands. Per exemple:

- x + y
- (x + y) * z
- x + y == z * 3 and not b

En la notació infixa cal usar parèntesis per desfer les ambigüitats, ja que `x + y * 3` podria significar `( x + y ) * 3` o bé `x + ( y * 3 )`. El segon cas pot prescindir dels parèntesis doncs la multiplicació és més prioritària que la suma.

En la **notació postfixa**, tal com el seu nom indica, l'operador va darrera els dos operands. Té l'avantatge de no necessitar parèntesis. Per exemple:

Infixa	Postfixa
x + y	x y +
(x + y) * z	x y + z *
x + y == z * 3 and not b	x y + z 3 * == b not and

Finalment, la **notació prefixa** consisteix en posar l'operador davant els dos operands i tampoc necessita parèntesis. Per exemple:

Infixa	Prefixa
x + y	+ x y
(x + y) * z	* + x y z
x + y == z * 3 and not b	and == + x y * z 3 not b

La nostra calculadora només treballarà amb expressions escrites amb notació infixa i postfixa.

1.2 Com guardar una expressió

Per poder manipular les expressions i avaluar-les, les emmagatzemarem en un arbre. Donat que hi ha operadors unaris (el `not`) i binaris (la resta), usarem un arbre binari però en el cas dels operadors unaris el fill dret no el tindrem en compte. En el nostre cas, hi haurà nodes sense fills (les fulles on hi haurà els operands: números, booleans, variables, etc.), nodes amb dos fills però només importarà el fill esquerre (l'operació `not`) i nodes amb dos fills (les altres operacions). Cal tenir en compte que l'operador `-` és la resta, una operació binària i no pas l'operador unari de canvi de signe. El signe dels operands enters el desarem dins dels propis enters.

Per exemple, l'expressió `x + y == z * 3 and not b` es representaria amb l'arbre:

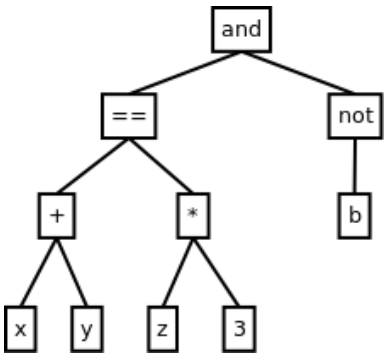

Figura 1: Arbre que guarda una expressió

Cadascun dels elements de l'arbre, que pot ser un operador o un operand, s'anomena token (element o peça en anglès).

Per construir un programa que sigui una calculadora d'expressions amb memòries, cal abans construir funcions capaces de manipular els tokens, també de llegir, escriure i avaluar expressions i finalment de poder guardar, consultar i eliminar expressions en diferents memòries. Gràcies als nostres coneixements de la programació orientada a objectes, resoldrem el problema definint vàries classes:

- **token** per gestionar els elements de les expressions: Operands i operadors
- **expressio** que permet llegir, escriure i avaluar expressions aritmètiques i booleanes
- **calculadora** que permet guardar, consultar i eliminar expressions en diferents memòries, així com avaluar-les individualment o en cascada

També farem ús de la classe **arbreBin** que implementa un arbre binari genèric o templatitzat.

2 Classe token

La classe `token` guarda un element d'una expressió:

- Operadors: `not`, `and`, `or`, `==`, `!=`, `+`, `-`, `*`, `/`, `%`, `**`
- Operands: Números enters (tant positius com negatius), constants booleanes (`T`, `F`) i variables.

i disposa dels següents mètodes públics:

```
class token {
  public:
    token();
    // Pre: true
    // Post: Crea un token enter de valor 0

    token(int i);
    // Pre: i = I
    // Post: Crea un token a partir de l'enter I

    token(bool b);
    // Pre: b = B
    // Post: Crea un token a partir del booleà B

    token(string s);
    // Pre: s = S
    // Post: Crea un token a partir del string S (descriu un operador o un operand)

    token(const char s[]);
    // Pre: s = S
    // Post: Crea un token a partir del vector de caràcters S (descriu un operador o un opera

    token(const token &t);
    // Pre: t = TK
    // Post: Crea un token a partir del token TK (constructor per còpia)

    token& operator=(const token &e);
    // Pre: t = TK
    // Post: Al p.i. se li ha assignat el token TK (operador assignació)

    ~token();
    // Pre: True
    // Post: S'ha destruit el token del p.i. (destructora)

    bool es_operador_unari() const;
    // Pre: true
    // Post: Indica si el p.i. és un operador unari

    bool es_operador_binari() const;
```

```
// Pre: true
// Post: Indica si el p.i. és un operador binari

bool es_operador_commutatiu() const;
// Pre: true
// Post: Indica si el p.i. és un operador commutatiu

bool es_boolea() const;
// Pre: true
// Post: Indica si el p.i. és una constant booleana

bool es_enter() const;
// Pre: true
// Post: Indica si el p.i. és una constant entera

bool es_variable() const;
// Pre: true
// Post: Indica si el p.i. és una variable

pair<int, bool> prioritat() const;
// Pre: true
// Post: Indica el nivell de prioritat a 'first' i si l'associativitat és
//       d'esquerra a dreta a 'second' del p.i.
//       Considerem que els operands tenen el màxim nivell de prioritat

int to_int() const;
// Pre: El p.i. és un token número enter
// Post: Retorna el valor enter del p.i.

bool to_bool() const;
// Pre:  El p.i. és un token booleà
// Post: Retorna el valor booleà true o false del p.i.

string to_string() const;
// Pre: true
// Post: Retorna el valor del p.i. convertit en string

bool operator==(const token &t) const;
// Pre: t = TK
// Post: Indica si el token del p.i. és igual al token TK

bool operator!=(const token &t) const;
// Pre: t = TK
// Post: Indica si el token del p.i. és diferent al token TK

friend istream& operator>>(istream& is, token &t);
// Pre: El canal is conté un string amb un contingut d'un token TK
// Post: t = TK
```

```
    friend ostream& operator<<(ostream& os, const token &t);
    // Pre: t = TK
    // Post: S'ha escrit al canal os el contingut del token TK */

  private:
    // Cal definir els atributs i mètodes privats dins del fitxer .rep
    #include "token.rep"
};
```

2.1 Aclariments

La següent taula mostra les prioritats i associativitats dels operadors usats en la pràctica:

Prioritat	**Operador**	**Associativitat**
7	**	De dreta a esquerra
6	* / %	D'esquerra a dreta
5	+ -	D'esquerra a dreta
4	== !=	D'esquerra a dreta
3	not	De dreta a esquerra
2	and	D'esquerra a dreta
1	or	D'esquerra a dreta

2.2 Tasques a fer

Cal definir els atributs i mètodes privats dins del fitxer `token.rep` i implementar els mètodes en el fitxer `token.cpp`. Disposeu de l'especificació de la classe a `token.hpp` i d'un programa principal `test_token.cpp` que testeja la majoria dels mètodes de la classe. També disposeu d'un fitxer `Makefile` per poder compilar i testejar la classe amb comoditat.

Per compilar:

```
make test_token.exe
```

Per testejar usant les dades d'entrada del fitxer `test_token.inp` i comparant la sortida amb la del fitxer `test_token.cor` que conté la sortida correcta:

```
make test_token
```

No us oblideu de fer els vostres programes i jocs de prova addicionals per testejar tots els mètodes de la classe.

3 Classe expressio

La classe `expressio` permet guardar i avaluar una expressió aritmètica o booleana i disposa dels següents mètodes públics:

```
class expressio {
  public:
  expressio(const token &t);
  // Pre: t = TK és un token operand
  // Post: Crea una expressió formada per operand TK

  expressio(const token &t, const expressio &e);
  // Pre: t = TK és un token operador unari, e = E
  // Post: Crea una expressió formada per l'operador unari TK aplicat a l'expressió E

  expressio(const token &t, const expressio &e1, const expressio &e2);
  // Pre: t = TK és un token operador binari, e1 = E1, e2 = E2
  // Post: Crea una expressió formada per l'operador binari TK aplicat a les expr. E1 i E2

  expressio(const expressio &e);
  // Pre: e = E
  // Post: Crea una expressió a partir de l'expressió E (constructor per còpia)

  expressio& operator=(const expressio &e);
  // Pre: e = E
  // Post: Al p.i. se li ha assignat l'expressio E (operador assignació)

  ~expressio();
  // Pre: True
  // Post: S'ha destruit l'expressió del p.i. (destructora)

  token arrel() const;
  // Pre: True
  // Post: Retorna el token de l'arrel del p.i.

  expressio fe() const;
  // Pre: L'arrel del p.i. és un operador
  // Post: Retorna l'expressió de l'esquerra del p.i.

  expressio fd() const;
  // Pre: L'arrel del p.i. és un operador binari
  // Post: Retorna l'expressió de la dreta del p.i.

  bool es_operand() const;
  // Pre: True
  // Post: Retorna si l'expressió del p.i. és un operand
  //   (o sigui que l'arbre binari que conté l'expressió és una fulla)

  list<token> operands() const;
```

```
// Pre: True
// Post: Retorna llista dels tokens operands de l'expressió del p.i., d'esquerra a dreta

void llegir_infixa(istream& is);
// Pre: El canal is conté una expressió en notació infixa i sense errors.
// Post: El p.i. conté l'expressió llegida del canal is.

void llegir_postfixa(istream& is);
// Pre: El canal is conté una expressió en notació postfixa i sense errors.
// Post: El p.i. conté l'expressió llegida del canal is.

string infixa() const;
// Pre: True
// Post: Retorna un string que conté l'expressió del p.i. amb notació infixa
//   amb el mínim nombre de parèntesis, cada element separat amb un espai

string postfixa() const;
// Pre: True
// Post: Retorna un string que conté l'expressió del p.i. amb notació postfixa,
//   cada element separat amb un espai

static expressio avalua_operador_unari(token op, expressio e);
// Pre: op = OP és un operador unari, e = E
// Post: Retorna l'expressió resultat d'avaluar l'operador OP sobre l'expressió E

static expressio avalua_operador_boolea(token op, expressio e1, expressio e2);
// Pre: op = OP és un operador binari booleà, e1 = E1, e2 = E2
// Post: Retorna expressió resultat d'avaluar l'operador OP sobre les expressions E1 i E2

static expressio avalua_operador_comparacio(token op, expressio e1, expressio e2);
// Pre: op = OP és un operador binari de comparació, e1 = E1, e2 = E2
// Post: Retorna expressió resultat d'avaluar l'operador OP sobre les expressions E1 i E2

static expressio avalua_operador_aritmetic(token op, expressio e1, expressio e2);
// Pre: op = OP és un operador binari aritmètic, e1 = E1, e2 = E2
// Post: Retorna expressió resultat d'avaluar l'operador OP sobre les expressions E1 i E2

expressio avalua() const;
// Pre: True
// Post: Retorna l'expressió resultant d'avaluar l'expressió del p.i. tot el que es pugui

expressio expandeix(token t, const expressio &e) const;
// Pre: t = TK és un token operand, e = E
// Post: Retorna l'expressió resultant de canviar tots els tokens TK
//       de l'expressió del p.i. per l'expressió E

private:
  // Cal definir els atributs i mètodes privats dins del fitxer .rep
```

```
    #include "expressio.rep"
};
```

3.1 Aclariments i consells

Una expressió, a diferència d'un arbre binari, mai pot ser buida. L'expressió més petita que podem tenir conté un únic operand (número enter, booleà o variable). Per aquest motiu la classe `expressio` no disposa del constructor d'expressió buida.

Per implementar aquesta classe us aconsellem que programeu els mètodes en ordre de dificultat creixent: Primer els mètodes bàsics (constructors, destructor, assignació, `arrel()`, `fe()`, `fd()`, `es_operand()`, `operands()`), després la lectura i escriptura d'expressions en notació postfixa, posteriorment l'avaluació d'expressions i finalment la lectura i escriptura d'expressions en notació infixa.

3.1.1 Lectura d'una expressió en notació postfixa

Per llegir una expressió en notació postfixa cal usar una pila amb els arbres parcials ja construïts. Cada cop que es llegeix un operador, es desapilen els operands necessaris i es construeix el nou subarbre parcial que es torna a apilar. Aquest algorisme és similar a l'usat en el problema del `Jutge`: Notació polonesa inversa (P52023), us recomanem que el resolgueu abans d'implementar la lectura d'una expressió postfixa.

Per marcar la fi de la lectura d'una expressió, tant en notació postfixa com infixa, al canal d'entrada o bé hi apareix l'string `->`, o bé no hi ha més dades per llegir.

3.1.2 Avaluació d'expressions

A continuació s'indiquen els casos a tenir en compte per avaluar diferents tipus d'operadors.

```
static expressio avalua_operador_unari(token op, expressio e);
// Pre: op = OP és un operador unari, e = E
// Post: Retorna l'expressió resultat d'avaluar l'operador OP sobre l'expressió E
```

Aquest mètode ha d'avaluar el cas:

- Operació `not` sobre un operand que és una constant booleana (`T`, `F`), el resultat és el càlcul pertinent (la seva negació).

En els altres casos es retorna l'arbre sense avaluar.

```
static expressio avalua_operador_boolea(token op, expressio e1, expressio e2);
// Pre: op = OP és un operador binari booleà, e1 = E1, e2 = E2
// Post: Retorna expressió resultat d'avaluar l'operador OP sobre les expressions E1 i E2
```

Aquest mètode ha d'avaluar els casos:

- Operació `and` on un dels dos operands és `F`, el resultat és `F`.

- Operació `or` on un dels dos operands és `T`, el resultat és `T`.
- Operació `and` on un dels dos operands és `T`, el resultat és l'altre operand.
- Operació `or` on un dels dos operands és `F`, el resultat és l'altre operand.
- Operació `and` o `or` entre dues constants booleanes (`T`, `F`), el resultat és el càlcul pertinent.

En els casos que no siguin cap dels anteriors es retorna l'arbre sense avaluar.

```
static expressio avalua_operador_comparacio(token op, expressio e1, expressio e2);
// Pre: op = OP és un operador binari de comparació, e1 = E1, e2 = E2
// Post: Retorna expressió resultat d'avaluar l'operador OP sobre les expressions E1 i E2
```

Aquest mètode ha d'avaluar el cas:

- Operació `==` o `!=` entre dues constants numèriques o dues constants booleanes (`T`, `F`) o dues variables el resultat és el càlcul pertinent (veure si son iguals o diferents, segons s'escaigui).

En els casos que no siguin cap dels anteriors es retorna l'arbre sense avaluar.

Heu de tenir en compte que l'operació `==` o `!=` entre dues variables només s'avalua si són la mateixa variable. Si les variables són diferents no es pot avaluar.

```
static expressio avalua_operador_aritmetic(token op, expressio e1, expressio e2);
// Pre: op = OP és un operador binari aritmètic, e1 = E1, e2 = E2
// Post: Retorna expressió resultat d'avaluar l'operador OP sobre les expressions E1 i E2
```

Aquest mètode ha d'avaluar els casos:

- Operació `*` on els un dels dos operands és zero, el resultat és zero.
- Operació `*` on els un dels dos operands és 1, el resultat és l'altre operand.
- Operació `+` on els un dels dos operands és zero, el resultat és l'altre operand.
- Operació `-` on el segon operand és zero, el resultat és el primer operand.
- Operació `/` on el primer operand és zero, el resultat és zero.
- Operació `/` on el segon operand és 1, el resultat és el primer operand.
- Operació `%` on el primer operand és zero, el resultat és zero.
- Operació `%` on el segon operand és 1, el resultat és zero.
- Operació `**` on el primer operand és zero, el resultat és zero.
- Operació `**` on el primer operand és 1, el resultat és 1.
- Operació `**` on el segon operand és zero, el resultat és 1.
- Operació `**` on el segon operand és 1, el resultat és el primer operand.
- Qualsevol de les quatre operacions sobre dos operands que siguin constants numèriques, el resultat és el càlcul pertinent.

En els casos que no siguin cap dels anteriors es retorna l'arbre sense avaluar.

```
expressio avalua() const;
// Pre: True
// Post: Retorna l'expressió resultant d'avaluar l'expressió del p.i. tot el que es pugui
```

Aquest mètode ha d'avaluar tot l'arbre, cridant al mètode adequat dels programats anteriorment per avaluar cada operació.

3.1.3 Lectura d'una expressió en notació infixa

Per llegir una expressió infixa cal usar una pila per guardar els operadors i els parèntesis d'obertura (poden ser tokens que contenen l'operador "(") i una pila d'arbres de tokens per guardar fragments de l'expressió reconstruïts. Durant la lectura caldrà tenir en compte la prioritat i l'associativitat de cada operador.

Funcionament de l'algorisme:

1. 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 (:
     - Per a cada operador crearem un arbre binari amb l'operador com arrel i un o dos arbres binaris de la pila d'arbres com a fills. Els arbres binaris usats els desapilarem. A continuació apilarem l'arbre binari creat a la pila d'arbres.
   - Si és un operand, crearem un arbre binari amb l'operand com arrel i l'apilarem a la pila d'arbres.
   - 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, desapilarem l'operador de la pila i crearem un arbre binari amb l'operador com arrel i un o dos arbres binaris de la pila d'arbres com a fills, els arbres binaris usats els desapilarem i a continuació apilarem l'arbre binari creat a la pila d'arbres.
     - Sempre al final hem d'apilar l'operador d'entrada a la pila d'operadors.
2. Quan s'acabi de llegir l'expressió infixa, cal desapilar tots els operadors que quedin fent cas omís dels parèntesis:
   - Per a cada operador, crearem un arbre binari amb l'operador com arrel i un o dos arbres binaris de la pila d'arbres com a fills. Els arbres binaris usats els desapilarem. A continuació apilarem l'arbre binari creat a la pila d'arbres.
3. Finalment dins de la pila d'arbres de tokens tindrem un únic arbre amb l'expressió complerta reconstruïda.

La següent figura mostra pas a pas com es reconstruiria l'expressió `( 2 + 5 ) * 3 + 1 ->`. En el nostre problema l'string `->` actua com a EOL (End Of Line): l'string `->` serveix per

marcar la fi de la lectura d'una expressió. A la part superior de la figura es veu l'evolució de la pila d'operadors i parèntesis d'obertura, a la part inferior l'evolució de la pila d'arbres de tokens per guardar fragments de l'expressió reconstruïts.

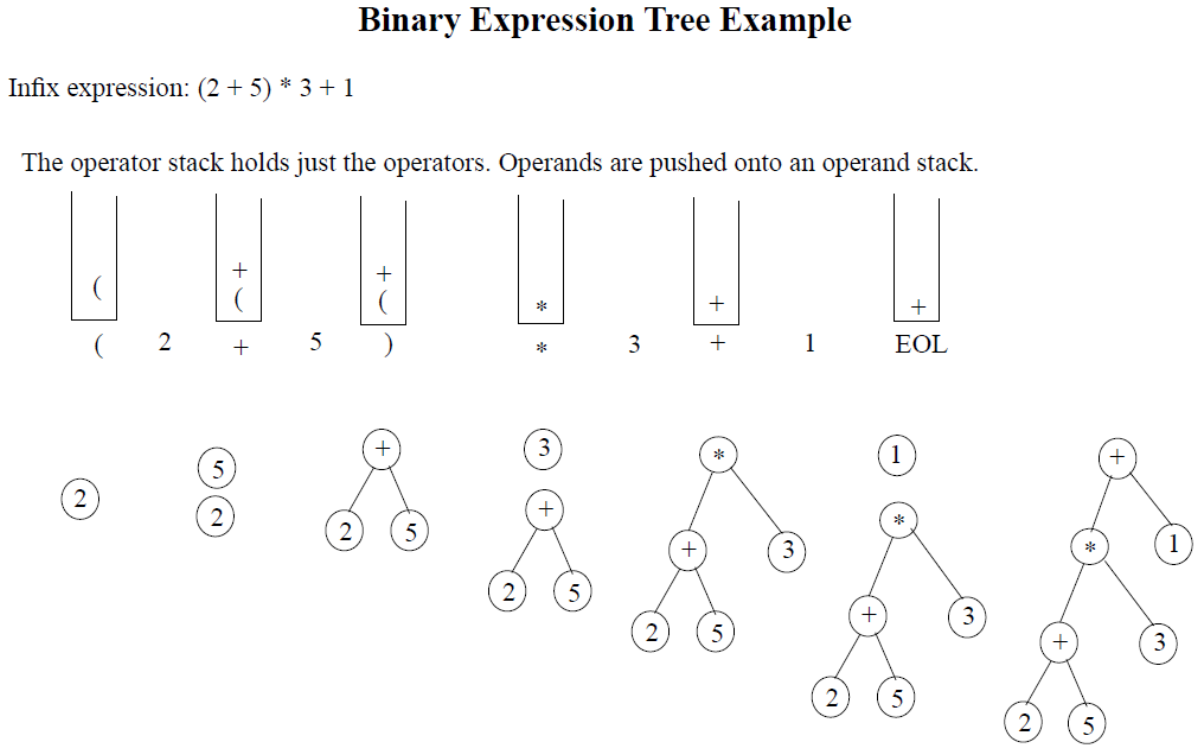

Figura 2: Reconstrucció d'un arbre a partir d'una notació infixa

Referències:

Shunting-yard algorithm: http://en.wikipedia.org/wiki/Shunting-yard_algorithm

3.1.4 Escriptura d'una expressió en notació infixa amb el mínim nombre de parèntesis

Podem fer una primera aproximació de com escriure una expressió en notació infixa fent un recorregut en inordre de l'arbre binari que conté l'expressió i afegir parèntesis d'obertura i tancament a les expressions que pengen de tots els operadors. Segurament molts dels parèntesis afegits són innecessaris.

Per a que l'expressió tingui el mínim nombre de parèntesis, per cada token operador OP:

- Només hem d'afegir parèntesis a l'expressió del fill esquerre si la prioritat del fill esquerre és menor que la d'OP o si tenen la mateixa prioritat i OP no és un operador commutatiu i l'associativitat d'OP és de dreta a esquerra.
- Només hem d'afegir parèntesis a l'expressió del fill dret si la prioritat del fill dret és menor que la d'OP o si tenen la mateixa prioritat i OP no és un operador commutatiu i l'associativitat d'OP és d'esquerra a dreta.

3.2 Tasques a fer

Cal definir els atributs i mètodes privats dins del fitxer `expressio.rep` i implementar els mètodes en el fitxer `expressio.cpp`. Disposeu de l'especificació de la classe a `expressio.hpp` i d'un programa principal `test_expressio.cpp` que testeja la majoria dels mètodes de la classe. També disposeu d'un fitxer `Makefile` per poder compilar i testejar la classe amb comoditat.

Per compilar:

```
make test_expressio.exe
```

Per testejar usant les dades d'entrada del fitxer `test_expressio.inp` i comparant la sortida amb la del fitxer `test_expressio.cor` que conté la sortida correcta:

```
make test_expressio
```

El programa principal llegeix una sèrie de comandes i les executa. Cada comanda és una línia amb el format següent:

```
FORMAT1 expressió -> FORMAT2
```

- `FORMAT1` pot ser `INFIXA` o `POSTFIXA` i indica la notació en la que cal llegir l'expressió subsegüent.
- `FORMAT2` pot ser `INFIXA` o `POSTFIXA` i indica la notació en la que cal escriure l'expressió un cop avaluada. També pot ser `OPERANDS`, en aquest cas escriu la llista dels operands de l'expressió un cop avaluada, o `EXPANDEIX`, en aquest cas llegeix un token T i una segona expressió E2 en notació infixa i escriu la primera expressió en notació infixa un cop s'ha expandit els seus tokens T amb l'expressió E2 i s'ha avaluat l'expressió resultant.

Per tant el programa llegeix cada línia, carrega l'expressió en el format `FORMAT1`, l'avalua i escriu una línia amb el resultat segons el format `FORMAT2`. També hi poden haver línies amb comentaris, comencen amb //, en aquest cas s'escriu a la sortida el mateix comentari de l'entrada.

Per simplicitat suposarem que l'entrada sempre és correcta. És a dir, no ni ha errors de format i les expressions sempre estan ben construïdes. I que sempre hi ha com a mínim un espai en blanc entremig de cada element (operador, constant, variable o parèntesis) que forma part de l'expressió per facilitar la lectura.

No us oblideu de fer els vostres programes i jocs de prova addicionals per testejar tots els mètodes de la classe.

4 Classe calculadora

La classe `calculadora` permet guardar expressions en les seves 26 memòries anomenades de la 'a' a la 'z' i avaluar expressions que usin expressions de les seves 26 memòries si fan menció a variables anomenades de la 'a' a la 'z'. Funciona com una calculadora convencional, si per exemple introduïm l'expressió

`2 * ( 3 + 2 )` donaria `10` com a resultat.

I funciona com una calculadora avançada on podem guardar expressions en les seves 26 memòries i usar-les posteriorment. Per exemple si guardem:

```
a = b * ( c + b )
b = 2
c = 3
```

després podem usar-les per calcular altres expressions:

`a` donaria `10` com a resultat.

`( a - b ) / 2` donaria `4` com a resultat.

`( a - b ) / ( c - d )` donaria `8 / ( 3 - d )` com a resultat.

La classe `calculadora` disposa dels següents mètodes públics:

```
class calculadora {
  public:
  calculadora();
  // Pre: true
  // Post: Crea una calculadora amb totes les memòries buides

  calculadora(const calculadora &c);
  // Pre: c = C
  // Post: Crea una calculadora a partir de la calculadora C (constructor per còpia)

  calculadora& operator=(const calculadora &c);
  // Pre: c = C
  // Post: Al p.i. se li ha assignat la calculadora C (operador assignació)

  ~calculadora();
  // Pre: True
  // Post: S'ha destruit la calculadora del p.i. (destructora)

  bool es_buida(char v) const;
  // Pre: v = V és un caràcter entre 'a' i 'z'
  // Post: Indica si la memòria V conté una expressió formada per
  //       una única variable el nom de la qual és V

  void guarda(char v, const expressio &e);
  // Pre: v = V és un caràcter entre 'a' i 'z', e = E
  // Post: Guarda a la memòria V l'expressió E
```

```
  void elimina(char v);
  // Pre: v = V és un caràcter entre 'a' i 'z'
  // Post: Elimina l'expressió de la memòria V

  expressio mostra(char v) const;
  // Pre: v = V és un caràcter entre 'a' i 'z'
  // Post: Retorna l'expressió de la memòria V sense avaluar-la

  expressio expandeix(char v) const;
  // Pre: v = V és un caràcter entre 'a' i 'z'
  // Post: Retorna l'expressió de la memòria V expandint-la
  //       usant les memòries del p.i. tot el que es pugui
  //       Si es detecta una circularitat retorna una expressió
  //       amb sola variable de nom "ERROR_expressions_circulars"

  expressio expandeix(const expressio &e) const;
  // Pre: e = E
  // Post: Retorna una expressió fruit d'expandir l'expressió E
  //       usant les memòries del p.i. tot el que es pugui
  //       Si es detecta una circularitat retorna una expressió
  //       amb sola variable de nom "ERROR_expressions_circulars"

  expressio avalua(char v) const;
  // Pre: v = V és un caràcter entre 'a' i 'z'
  // Post: Retorna l'expressió de la memòria V expandint-la i avaluant-la
  //       usant les memòries del p.i. tot el que es pugui
  //       Si es detecta una circularitat retorna una expressió
  //       amb sola variable de nom "ERROR_expressions_circulars"

  expressio avalua(const expressio &e) const;
  // Pre: e = E
  // Post: Retorna una expressió fruit d'expandir i avaluar l'expressió E
  //       usant les memòries del p.i. tot el que es pugui
  //       Si es detecta una circularitat retorna una expressió
  //       amb sola variable de nom "ERROR_expressions_circulars"

  private:
    // Cal definir els atributs i mètodes privats dins del fitxer .rep
    #include "calculadora.rep"
};
```

4.1 Aclariments i consells

Fixeu-vos que la calculadora disposa de 26 memòries anomenades de la 'a' a la 'z' (són els caràcters que hi ha entre la 'a' i la 'z' en el codi ASCII; com a identificadors de memòria no s'usen vocals accentuades ni caràcters inexistents en l'alfabet anglès com la ç o la ñ).

La calculadora constantment ha d'accedir a una de les 26 memòries per guardar o consultar expressions, per tant cal que l'accés a una memòria concreta sigui ràpid (una cerca lineal, encara que la calculadora tingui 26 memòries, seria ineficient).

El fet que una memòria de la calculadora estigui buida implica que l'expressió que conté sigui una expressió formada per una única variable el nom de la qual coincideix amb el nom de la memòria buida. Si per exemple la memòria 'a' està buida, al consultar la memòria 'a' ens retornaria l'expressió `a`.

La nostra calculadora llegirà i escriurà totes les expressions en notació infixa. Les expressions amb notació infixa de la nostra calculadora poden acabar amb l'string `->` o perquè al canal d'entrada (la línia) no hi ha més dades.

Els mètodes de la classe `calculadora` permeten realitzar tres nivells de consulta de les memòries de la calculadora:

- `mostra`: Mostra l'expressió que hi ha a la memòria tal qual.
- `expandeix`: Expandeix les variables (de la 'a' a la 'z') de l'expressió que hi ha a la memòria consultant expressions d'altres memòries i substituint-les, repetint el procés tot el que es pugui.
- `avalua`: Expandeix les variables (de la 'a' a la 'z') de l'expressió que hi ha a la memòria consultant expressions d'altres memòries i substituint-les, repetint el procés tot el que es pugui. Finalment s'avalua l'expressió resultant.

Per exemple si guardem a les memòries `a`, `b` i `c` les següents expressions:

```
a = b * 2
b = c + 1
c = 3
```

i consultem la memòria `a` amb els tres nivells de consulta que ens ofereix la nostra calculadora obtindríem com a resultat aquestes expressions:

- `mostra a`: `b * 2`
- `expandeix a`: `( 3 + 1 ) * 2`
- `avalua a`: `8`

Quan expandim una expressió o una memòria podria succeir que les expressions estiguessin relacionades entre elles de forma circular de forma que podríem entrar en un bucle o crida recursiva sense fi. Hem de ser capaços de detectar quan l'expressió d'una memòria ja s'està expandint i no expandir-la una segona vegada. Per exemple si guardem a les memòries `a`, `b` i `c` les següents expressions:

```
a = b * 2
b = c + 1
c = a
```

i expandim o avaluem la memòria `a` el procés seria el següent (i no acabaria mai):

```
b * 2
( c + 1 ) * 2
( a + 1 ) * 2
( b * 2 + 1 ) * 2
( ( c + 1 ) * 2 + 1 ) * 2
( ( a + 1 ) * 2 + 1 ) * 2
( ( b * 2 + 1 ) * 2 + 1 ) * 2
...
```

En cas de detectar una expansió d'expressions circular, la nostra calculadora retornarà una expressió formada per una única variable anomenada "ERROR_expressions_circulars". Això normalment s'implementa usant la gestió d'errors que ofereix el llenguatge de programació (en el cas del llenguatge C++ les instruccions `throw` i `catch`), però això ho veureu amb detall a l'assignatura ESIN.

4.2 Tasques a fer

Cal definir els atributs i mètodes privats dins del fitxer `calculadora.rep` i implementar els mètodes en el fitxer `calculadora.cpp`. Disposeu de l'especificació de la classe a `calculadora.hpp` i d'un programa principal `test_calculadora.cpp` que testeja la majoria dels mètodes de la classe. També disposeu d'un fitxer `Makefile` per poder compilar i testejar la classe amb comoditat.

Per compilar:

```
make test_calculadora.exe
```

Per testejar usant les dades d'entrada del fitxer `test_calculadora.inp` i comparant la sortida amb la del fitxer `test_calculadora.cor` que conté la sortida correcta:

```
make test_calculadora
```

El programa principal llegeix una sèrie de comandes, una per cada línia, les executa i escriu una línia amb el resultat. Cada comanda és una línia amb un dels següents formats:

- `// comentari`: Línia que conté un comentari, s'escriu el comentari a la sortida.
- `exp`: Avalua l'expressió `exp` i l'escriu a la sortida
- `m = exp`: Guarda a la memòria `m` l'expressió `exp` i la mostra tal qual a la sortida
- `m <<`: Elimina l'expressió de la memòria `m` i escriu a la sortida l'expressió que queda a la memòria `m`
- `m >>`: Mostra l'expressió de la memòria `m` tal qual a la sortida
- `m >>>`: Expandeix l'expressió de la memòria `m` i l'escriu a la sortida
- `m`: Avalua l'expressió de la memòria `m` i l'escriu a la sortida

Per simplicitat suposarem que l'entrada sempre és correcta. És a dir, no ni ha errors de format i les expressions sempre estan ben construïdes. I que sempre hi ha com a mínim

un espai en blanc entremig de cada element (operador, constant, variable o parèntesis) que forma part de l'expressió per facilitar la lectura.

No us oblideu de fer els vostres programes i jocs de prova addicionals per testejar tots els mètodes de la classe.

A Estructura de fitxers

Els fitxers per desenvolupar la pràctica són els següents:

`token.hpp`	Definició dels mètodes públics de la classe `token`.
`expressio.hpp`	Definició dels mètodes públics de la classe `expressio`.
`calculadora.hpp`	Definició dels mètodes públics de la classe `calculadora`.
`token.rep`	Definició dels atributs i mètodes privats de la classe `token`.
`expressio.rep`	Definició dels atributs i mètodes privats de la classe `expressio`.
`calculadora.rep`	Definició dels atributs i mètodes privats de la classe `calculadora`.
`token.cpp`	Implementació dels mètodes de la classe `token`.
`expressio.cpp`	Implementació dels mètodes de la classe `expressio`.
`calculadora.cpp`	Implementació dels mètodes de la classe `calculadora`.

Tots ells estan disponibles a `/home/public/pro1/practica1920Q2`. En el mateix directori hi ha els següents programes principals per testejar que hem programat correctament les classes anteriors i un `Makefile` per compilar-los i testejar-los:

`test_token.cpp`	Programa de prova de la classe `token`.
`test_expressio.cpp`	Programa de prova de la classe `expressio`.
`test_calculadora.cpp`	Programa de prova de la classe `calculadora`.

B Lliuraments

- Tots els lliuraments s'han de fer pel campus digital.
- Cada lliuraments consistirà en **un únic** fitxer comprimit, amb nom `dni1-dni2.zip` format pels DNIs dels membres del grup, ordenats de menor a major. El format de compressió ha de ser .zip. Si el nom del fitxer no compleix aquest format, pot significar la qualificació de NO PRESENTAT per al lliurament.
- Aquest fitxer contindrà tots els fitxers del lliurament, sense cap altre subdirectori addicional.
- Cada lliurament ha de contenir:
  - *Tots* els fitxers de codi de la llista corresponent (veure més avall). El codi ha d'estar comentat descrivint quin és el resultat de cada funció (postcondició), i en quines condicions és aplicable (precondició). Cal definir la hipòtesis d'inducció (HI) i la funció de fita (FF) de totes les crides recursives.
  - Un fitxer de **text pla** (.txt) amb el nom, cognoms i DNI dels membres de l'equip.

  Un paquet incomplet pot significar la qualificació de NO PRESENTAT per al lliurament.

- **Primer lliurament:**
  - *Data límit:* 17 de maig del 2020 a les 23:55h.
  - *Codi a presentar:*

Definicions classes	*Implementacions classes*
`token.rep`	`token.cpp`
`expressio.rep`	`expressio.cpp`

- **Segon lliurament:**
  - *Data límit:* 7 de juny del 2020 a les 23:55h.
  - *Codi a presentar:*

Definicions classes	*Implementacions classes*
`token.rep`	`token.cpp`
`expressio.rep`	`expressio.cpp`
`calculadora.rep`	`calculadora.cpp`