LCPL

October 27, 2015

1 Descriere generală

1.1 Structura unui program LCPL

Tot codul LCPL este organizat în **clase**, similar cu Java. Un program LCPL trebuie definit în întregime într-un fișier. Un fișier poate conține mai multe clase.

Definiția unei clase este (`[...]` reprezintă construcții opționale):

```
class <nume> [inherits <nume>]
  <membri>
end;
```

Clasele conțin zero sau mai mulți `membri`, ce pot fi `atribute` sau `metode`.

Un atribut reprezintă date interne clasei și nu poate fi accesat direct decât din interiorul clasei. Pentru accesul din exterior, trebuiesc folosite metodele.

Declarația unui atribut are forma:

```
<tip> <nume> [ = <expresie> ] ;
```

Atributele unei clase se declară într-o secțiune `var ... end;`. Pot exista mai multe asemenea secțiuni într-o clasă. Atributele definite într-o secțiune sunt vizibile pe toată lungimea clasei (chiar și înainte de apariția în text a secțiunii).

```
var Int xcar; List xcdr; end;
```

Fiecare atribut are un tip care trebuie declarat explicit de programator. În exemplul de mai sus, tipul lui `xcar` este `Int` iar tipul lui `xcdr` este `List`.

Un atribut poate fi declarat împreună cu o inițializare:

```
var Int xcar = 2 + 3; end;
```

În acest caz, este o eroare dacă tipul expresiei cu care se inițializează atributul nu corespunde cu tipul declarat. Expresia cu care se inițializează un atribut poate fi orice expresie permisă în limbaj. Se pot apela metode, se pot referi alte atribute ale clasei respective, se pot folosi bucle `while` sau expresii condiționale `if`, ca în exemplul următor:

```
var
  Shape s = [baseShape];
  Vertex vx =
    if [s.sides] == 3 then
      local
        Vertex vx = new Vertex;
        Vertex vy;
      end;
      [vx.init 0,0,null];
      vy = new Vertex;
      [vy.init 0,2,vx];
      vx = new Vertex;
      [vx.init 1,1,vy];
      vx;
    else
      if 1 == [s.sides] then
        local
          Vertex vx = new Vertex;
          Vertex vy = new Vertex;
        end;
        [vx.init 1,1,null];
        [vy.init 0,0,vx];
        vy;
      else
        null;
      end;
```

```
    end;
end;
```

O metodă este de forma:

```
<nume> [ <argumente> ]  [ -> <tip> ] : <corp> end;
```

Tipul unei metode reprezintă lista argumentelor acesteia - separate prin caracterul `,` - precum și tipul expresiei întoarse de metodă (prima metoda din exemplul de mai jos). În cazul în care metoda nu întoarce o expresie, tipul întors lipsește (a doua metodă). În cazul în care o metodă nu are argumente, lista lor lipsește (ultima metodă).

```
hello Int a, String b -> String :
  "Hello " + b + a;
end;

printHello Int a, String b :
   [out [hello a,b]];
end;

piTimes100 -> Int :
   314;
end;
```

Este posibil să existe și metode fără argumente și fără tip întors, de exemplu metoda următoare ce conține un corp vid:

```
empty: end;
```

Un program LCPL trebuie sa contina o clasa `Main` cu o metodă `main`, fără argumente, definită în această clasă, sau moștenită din părinții clasei `Main`. La pornirea programului se va crea un obiect de tip `Main`, se va initializa, și apoi se va apela metoda `main`.

LCPL este un limbaj case sensitive, `Cons` și `cons` reprezintă două lucruri diferite.

1.2 Tipuri de date și clase

Tipurile de date din LCPL sunt numerele întregi (`Int`) și clasele.

O clasă poate moșteni de la o singură superclasă (folosind `inherits` urmat de numele clasei moștenite).

Numele unei clase este public vizibil în tot programul, nu există ierarhii de module. Prin urmare, este o eroare să aveți două clase cu același nume (nu puteți redefini o clasă).

Este o eroare definirea a două atribute sau a două metode cu același nume în cadrul aceleiași clase, dar este perfect legal cazul în care un atribut și o metodă au același nume.

Între clase există o ierarhie de tipuri creată pe baza relațiilor de moștenire. O clasă moștenește o singură altă clasă; în cazul în care nu este declarată clasa părinte se va moșteni automat clasa specială `Object` de la rădăcina ierarhiei de clase.

Ierarhia de clase este de fapt un graf `aciclic` de clase. Moștenirea ciclică reprezintă o eroare. De asemenea, este ilegală moștenirea unei clase a cărei definiție lipsește din program.

Moștenirea presupune copierea tuturor membrilor clasei părinte în clasa copil. Din cauza restricțiilor de nume legate de membrii din aceeași clasă, nu se poate redefini un atribut. Redefinirea unei metode suprascrie metoda din clasa părinte, și este permisă cât timp se păstrează intacte tipurile argumentelor și tipul valorii întoarse.

Numele argumentelor unei metode trebuie să fie diferite. În cazul în care un argument are același nume cu un atribut al clasei, atributul este invizibil pe parcursul metodei.

Pentru a aloca spațiu pentru obiecte și a le putea folosi, se folosește operatorul `new`. Nu există mecanism pentru eliberarea spațiului ocupat de obiecte, LCPL are un management automat al memoriei.

La crearea unei instanțe a clasei folosind `new`, se vor inițializa *în ordinea definirii* toate atributele acesteia, începând cu atributele clasei părinte. Dacă nu există inițializare explicită, atunci se va folosi o inițializare implicită: exceptând clasele speciale prezentate mai jos, celelalte atribute se vor inițializa la valoarea `null` (echivalentul lui `null` din Java și al lui `NULL` din C/C++).

Valorile `null` pot fi asignate unei variabile și se pot face comparații pe ele. Apelarea de metode ale unei instanțe de obiect cu valoare `null` generează o eroare la runtime.

1.3 Metode

Corpul unei metode este format dintr-un bloc de `instrucțiuni`. Acestea pot fi simple expresii aritmetice, logice sau pe șiruri, instanțieri de obiecte, apeluri de metode, atribuiri și instrucțiuni de control. Toate instrucțiunile ce apar în corpul unei metode sunt terminate prin `;`.

```
class Main
    main:
        local Cons c; Int x; end;
        c = new Cons;
        x = 0;
    end;
end;
```

Exemplul de mai sus declară o variabilă `c` de tip `Cons` și un întreg `x`, locale metodei `main`. O metodă poate conține **oricâte** construcții `local ... end;` folosite pentru declararea de variabile locale metodei. Fiecare nume de variabilă local este vizibil din momentul apariției definirii, până la finalul blocului în care se află. Există posibilitatea ca o variabilă definită într-o construcție `local ... end;` să ascundă un nume de variabilă definit anterior (local, parametru sau atribut). Variabilele locale unei metode pot fi definite și inițializate, exact ca la atribute:

```
local Cons c = new Cons; Int x = 0; end;
```

Dacă o metodă întoarce o valoare, corpul acesteia trebuie să se termine cu o instrucțiune al cărei tip corespunde celui întors de metodă. De exemplu, următoarea funcție întoarce valoarea instrucțiunii `if` , de tipul `Int`.

```
fact Int n -> Int:
    if n < 1 then
        1;
    else
        n * [fact n - 1];
    end;
end;
```

Este o eroare cazul în care metoda întoarce o valoare de un tip dar are corpul vid, de exemplu:

```
fact Int n -> Int: end; # eroare
```

În cazul în care metoda nu declară nici un tip întors, se considera ca metoda întoarce tipul `Void`. Este o eroare să folosiți rezultatul întors de o astfel de metodă:

```
m Int n: n; end;
...
a = [m 42]; # eroare!!
```

Următorul exemplu ilustrează un program cu 2 clase:

```
class Cons
    var Int xcar; Cons xcdr; end;

    size -> Int:
        1 + if xcdr == null then 0; else [xcdr.size]; end;
    end;

    init Int hd, Cons tl:
        xcar = hd;
        xcdr = tl;
    end;
end;

class Main
    main:
        local Cons c; Int x; end;
        c = new Cons;
        x = 0;
        [c.init x, c];
    end;
end;
```

1.4 Expresii

Cele mai simple expresii din limbaj sunt **constantele**. Acestea pot fi întregi (valorile boolene fiind mapate pe principiul `0` - fals, altfel adevărat) și șiruri

de caractere. Tipul constantelor întregi este `Int` iar al celor de tip șir de caractere `String`.

Cuvantul cheie `null` este o constanta ce intoarce valoarea `null` .

Numele de variabile, parametrii formali, atributele și `self` sunt **identificatori**, deci expresii. Tipul fiecărei expresii de această formă este tipul cu care a fost declarat identificatorul.

Pentru a putea accesa membrii clasei din interiorul acesteia (util în cazul în care ei sunt ascunși de parametrii formali sau de variabile locale), se poate folosi `self` (exact ca `this` din C++ și Java).

Atributele sunt vizibile doar în interiorul clasei, pe toată durata acesteia. Variabilele locale sunt vizibile doar în interiorul metodei în care sunt definite, pornind de la punctul în care au fost definite.

O **atribuire** este o expresie de forma:

```
<id> = <expr>
```

Tipul întors de atribuire este tipul cu care a fost declarat `<id>`. Valoarea unei atribuiri este valoarea expresiei `<expr>`.

Cele două tipuri trebuie să corespundă, sau tipul expresiei `<expr>` să poată fi convertit implicit la tipul expresiei `<id>`. Se face conversia implicită de la `Int` către `<String>`. Expresia `<expr>` nu trebuie să aibă tipul `Void`.

Pentru o variabilă de un anumit tip se poate folosi orice tip aflat mai jos în ierarhia de clase. In exemplul de mai jos, unei variabile `Shape` i se poate atribui o expresie de tip `Circle` fără a fi nevoie de o conversie explicită.

```
class Shape
  print IO stream :
    [stream.out "This is a generic shape"];
  end;
end;

class Circle inherits Shape
  print IO stream :
    [stream.out "This is a circle"];
  end;
end;
```

Pe de altă parte, pentru conversia inversă trebuie să realizăm o conversie explicită folosind sintaxa { `tip expresie_de_convertit` }.

Conversia explicită către un alt tip, *cast*, este și ea o expresie. Tipul întors este tipul către care se face conversia. Dacă nu se poate face conversia se va genera o eroare la rulare.

Aceleași reguli de conversie a tipurilor se aplică și inițializărilor din construcțiile `local`, `var`, sau parametrilor la apelul unei metode.

Apelul unei metode, `dispatch`, se realizează folosind una din construcțiile:

```
[<expr>.<id> <expr>, ... <expr>]
[<expr>.<id>]
[<id> <expr>, ... <expr>]
[<id>]
```

Ultimele două cazuri sunt o scurtătură, forma `<id>` fiind de fapt `self.<id>`.

Prima formă reprezintă un apel de metodă cu argumente, fiecare argument fiind separat prin spațiu de celelalte. A doua formă este un apel de metodă fără argumente.

Tipul întors de o expresie de tip dispatch este tipul întors de metodă, sau tipul `Void`, daca metoda nu are declarat un tip întors. Valoarea unui apel de metodă este valoarea ultimei instrucțiuni executate de metoda respectiva.

Este o eroare să furnizați un număr diferit de argumente decât sunt necesare, sau să folosiți un argument de un tip necorespunzător.

LCPL suportă polimorfismul. In cazul in care o metodă este redefinită într-o clasă derivată, și o variabilă de tip bază referă un obiect de tip derivat, expresia dispatch va apela metoda din clasa derivată și nu cea din clasa de bază.

Dacă se doreste specificarea explicită a metodei care va fi apelată, fara a folosi mecanismul de polimorfism, se poate folosi sintaxa de dispatch static:

```
[<expr>::<tip>.<id> <expr>, ... <expr>]
[<expr>::<tip>.<id>]
```

De exemplu:

```
class Shape
  print IO stream :
    [stream.out "This is a generic shape"];
  end;
end;
```

```
class Circle inherits Shape
  print IO stream :
    [stream.out "This is a circle"];
  end;
end;

class Main inherits IO
  main :
    local
      Shape s = new Circle;
    end;
    [s.print self] ; # This is a circle
    [s::Shape.print self] ; # This is a generic shape
  end;
end;
```

Dacă se dorește dispatch static pentru o metodă a obiectului `self`, acesta trebuie specificat explicit, ca în exemplul de mai jos:

```
class Shape
  var Int x; Int y1; Int x2; Int y2; end;
  init Int x, Int y, Int dx, Int dy :
    self.x = x; self.y = y;
    x2 = x + dx; y2 = y + dy;
  end;
end;

class Ellipse inherits Shape
  var Int rx; Int ry; end;
  init Int x, Int y, Int dx, Int dy :
    [self::Shape.init x, y, dx, dy];
    rx = dx / 2; ry = dy / 2;
  end;
end;
```

O **expresie condițională** este de formele:

```
if <expr1> then <expr>;...<expr>; else <expr>;...<expr>; end
if <expr1> then <expr>;...<expr>; end
```

Tipul **if**-ului este tipul ultimei instrucțiuni din cele două ramuri, sau tipul `Void` în cazul în care ramura **else** nu există, una din ramuri nu întoarce nimic, sau tipurile întoarse de cele două ramuri nu sunt compatibile (unul din ele nu se poate converti către celălalt). Valoarea if-ului este valoarea ultimei instrucțiuni executate.

O **buclă** are forma

```
while <expr1> loop <expr>;...<expr>; end
```

O expresie **while** are tipul `Void` și nu întoarce nicio valoare.

Atât pentru `if` cât și pentru `while`, `expr1` reprezintă o expresie al cărei tip este întreg cu semnificația că orice valoarea nenulă înseamnă true, 0 înseamnă false.

În fiecare ramură de `if` și în interiorul buclei `while` se pot pune mai multe instrucțiuni. Blocurile de instrucțiuni respective pot conține construcții `local`. Variabilele declarate în acele construcții sunt vizibile până la sfarsitul expresiei condiționale sau al buclei.

Pentru a construi și inițializa un nou obiect se folosește `new`.

```
new <type>
```

Se întoarce obiectul nou construit, de tipul corespunzător.

Operațiile aritmetice (`+`, `-`, `*`, `/`), de comparație (`<`, `<=`) și de egalitate (`==`) sunt expresii. Exceptând `+` și `==`, ambii operanzi sunt întregi și valoarea întoarsă este un întreg (0 sau 1).

Daca operanzii lui `==` sunt `Int` sau `String` se compară conținutul celor doi operanzi. Un `Int` poate fi comparat cu un `String` prin conversie implicită la `String`. Altfel, dacă cei doi operanzi sunt clase derivate din `Object`, `==` întoarce "1" doar dacă referă același obiect, sau sunt ambii `null`. Operanzii nu trebuie sa aiba tipul `Void`.

Folosirea unui argument de tip `String` în cazul operatorului `+` forțează tipul celuilalt argument la `String`: dacă tipul argumentului e diferit de `Int` sau `String` se aruncă eroare de tip la compilare, altfel întregul e convertit la șir și șirul este lăsat nemodificat.

Operatorul `-` funcționează și ca operator unar, pentru a obține un număr negativ.

Pentru a nega un întreg (`0` $\rightarrow$ `1`, orice altceva $\rightarrow$ `0`) se folosește `!`.

Pentru a grupa expresii se pot folosi paranteze: `(` și `)`.

1.5 Prioritatea operatorilor și reguli de asociativitate

Următoarea listă listează operatorii, în ordinea descrescătoare a priorității lor:

```
.
[metoda] string[...] {...} if while
- unar
* /
+ -
< <= ==
!
= new
```

Operatorul `-` unar are precedență față de operatorul binar. Exceptând `=`, `<`, `<=` și `==`, toți operatorii binari sunt asociativi stânga. Operatorii de comparație nu sunt asociativi. Operatorul `=` este asociativ dreapta.

2 Clase speciale și valori întregi

LCPL are 3 clase speciale: `Object`, `IO` și `String`.

Întregii sunt un alt tip fundamental; `Int` nu este o clasă derivată din `Object`. Implicit, întregii se inițializează cu `0`.

2.1 `Object`

Clasa `Object` este radăcina ierarhiei de clase. Sunt definite următoarele metode:

```
abort
typeName -> String
copy -> Object
```

Metoda `abort` termină programul forțat.

Metoda `typeName` întoarce numele clasei originale a obiectului. Următoarele apeluri ale metodei `typeName` întorc o constantă de tip `String` cu valoarea `"Circle"`:

```
method_test:
    local Circle c; Shape s = c; Object o = c; end;
    [c.typeName];
    [s.typeName];
    [o.typeName];
    end;
```

Metoda `copy` realizează o copie de suprafață a obiectului, fără a copia și referințele acestuia.

Orice instanță a unei clase ce derivează din `Object`, inclusiv `Object` se va inițializa implicit cu `null`.

2.2 IO

Pentru a avea acces la standard input și output trebuiesc folosite metodele din clasa `IO`, prin intermediul unui obiect al acestui tip.

Metodele sunt:

```
out String msg -> IO
in -> String
```

Metoda `out` tipărește un șir pe standard output și întoarce obiectul de tip IO.

Metoda `in` citește standard input până la sfarșitul liniei și întoarce șirul citit, fără caracterul *new line*.

Este o eroare redefinirea clasei `IO`, dar aceasta poate fi derivată.

Inițializarea implicită se face tot prin `null`.

2.3 String

Clasa `String` reprezintă șirurile. Metodele ei sunt:

```
length -> Int
toInt -> Int
```

Metoda `toInt` poate fi folosită pentru a converti un șir la întreg. În cazul în care șirul pe care este aplicată metoda nu poate fi parsat ca un întreg (vezi exemplele) se va întoarce valoarea 0.

```
["42".toInt] # -> 42
["42a".toInt] # -> 0
["abc42".toInt] # -> 0
["abc".toInt] # -> 0
```

În plus, clasa `String` suportă o sintaxă specială pentru extragerea unui subșir dintr-un șir:

```
s[start, final]
```

în cazurile în care `start < 0`, `start > final` sau unul din indici este în afara șirului, se va genera o eroare la rulare. Indexarea începe de la `0` și extrage șirul de la `start` inclusiv până la `final` exclusiv.

```
"abcd"[0,4] # -> "abcd"
"abcd"[1,3] # -> "bc"
"abcd"[-1,3] # eroare
"abcd"[1,5] # eroare
"abcd"[2,2] # ""
"abcd"[3,2] # eroare
```

Mai mult, șirurile se pot concatena folosind `+`.

```
"as" + "df" # -> "asdf"
```

Un alt avantaj al clasei `String` este conversia implicită către `String` a valorilor întregi, în cazul folosirii unuia dintre operatorii `+` sau `==` precum și în cazul unei atribuiri către o variabilă sau parametru de tip `String`.

```
4 + "2"
"4" + 2
local String s = 42; end;
42 == "42"  # -> true
```

În final, inițializarea default a unui `String` este `""`.

3 Structură lexicală

LCPL conține următorii atomi lexicali:

întregi șiruri nevide de cifre `0 - 9` care sunt `0` sau nu încep cu `0`.

identificatori șiruri diferite de cuvintele cheie, conținând litere, cifre și _. Trebuie să înceapă cu o literă.

șiruri secvențe de caractere încadrate de `"..."`. În interiorul unui șir, secvența `\c` este `c` exceptând cazurile:

- `\n` linie nouă
- `\r` carriage return
- `\t` tab

Un șir trebuie să nu conțină un final de rând neescapat.

```
"This is a valid \
string."
"This is
not valid."
```

Toate șirurile dintr-un fișier trebuiesc terminate până la finalul lui.

comentariile sunt doar pe o singură linie, încep cu `#` și se termină la final de linie

```
# this is a comment # still in comment
```

Cuvinte cheie `class`, `inherits`, `end`, `var`, `local`, `null`, `new`, `if`, `then`, `else`, `while`, `loop`, `self`.

White space blank (ASCII 32), `\n` (newline), `\r` (carriage return), `\t` (tab).