Limbaje şi Translatoare

Se numeşte limbaj (de programare/comunicare/reprezentare/descriere, protocolar, lingvistic etc.) un set bine definit de simboluri, expresii şi reguli (lexicale, sintactice, semantice, pragmatice/semiotice) care permit descrierea sau transmiterea unor informaţii. Se numeşte translator un sistem / procedeu / program de conversie a unor elemente descrise într-un limbaj în elementele corespunzătoare unui alt limbaj. Translarea reprezintă conversia unor date codate într-un format în date codate în alte formate, a unor elemente din limbajul natural în limbaje/formate digitale, a unor elemente lingvistice, a unor informaţii aleatoriu distribuite în informaţii sistematizate (data/web mining). Lexicul reprezintă mulţimea atomilor/simbolurilor (tokeni). Sintaxa reprezintă regulile de combinare a simbolurilor lexicale. Semantica reprezintă sensul/înţelesul/atributul unor construcţii/combinaţii sintactico-lexicale (relaţiile dintre simboluri, obiecte şi reprezentări). Pragmatica reprezintă capacitatea de a caracteriza/descrie (limbaj) sau a transmite/converti (translator) fidel/eficient informaţia (relevanţa/corelaţia cod-mesaj). Semiotica reprezintă relaţia simboluri – obiecte – interpretări, Semiotica=Sintaxa+Semantica+Pragmatica.

Un limbaj/translator poate să prezinte în grade diferite o serie de caracteristici precum: extensibilitatea, compatibilitatea, eficienţa, portabilitatea, verificabilitatea, integritatea, modularitatea, fiabilitatea, ortogonalitatea, lizibilitatea, consistenţa, uniformitatea, demonstrabilitatea. Limbajele pot opera sau descrie structuri informativ corelate pe diferite niveluri. Nivel Uman/Matematic – modalitatea de descriere specific umană utilizând elemente naturale (lingvistice) sau logice (matematice). Nivel ridicat(înalt)/Simbolic (High-Level) – operarea cu elemente apropiate de limbajul natural dar adaptate/adresate unui alt sistem (digital, biologic etc.). Nivel coborât – Sistem/Maşină (Low-Level) – operarea cu elemente apropiate de operaţiile efectuate de sistem (Procesor/Maşină/Entitate biologică).

Programarea (descrierea/reprezentarea şi transformarea unor informaţii) este un proces de translare a unor intrări în ieşiri, proces intermediat de către un sistem de a cărui limbaj trebuie să ţinem cont în ultimă instanţă. Arhitectura unui sistem de calcul reprezintă modalitatea în care sunt distribuite şi interconectate elementele fundamentale necesare procesării. Arhitectura Princeton (von Neumann) reprezintă baza majorităţilor arhitecturilor, permite accesul complet/combinat/nerestrictiv la zona de date şi cod. În arhitectura Harvard zonele de cod şi date sunt gestionabile separat (spaţial) dar simultan (temporal). Aceste denumiri sunt utilizate şi pentru stilurile de combinare date-coduri.

1. A se propune/concepe şi analiza un limbaj care să conţină cel puţin operaţii de input-output şi
- algebrice. A se identifica elementele lexicale, sintactice şi semantice: simboluri, cuvinte rezervate, operatori, operanzi, instrucţiuni etc. A se implementa un program interpretor/translator/compilator care să interpreteze/compileze/proceseze/transleze limbajul/codul propus. Comenzile `Output`/`Input` pot fi considerate comunicări cu consola etc. A se identifica toate translările efectuate până la vizualizarea outputului de către un operator uman. A se adăuga limbajului noi elemente precum: macro-substituţii, alte operaţii aritmetice, binare sau logice, instrucţiuni de control, optimizări la nivelul translatorului etc. Exemplu de limbaj/cod ad-hoc de lucru, numit generic MyLang:

```
Var a = ?, b = 2, c = 3
a = b + c
Var d = 1
a = a + d
Output a
Input d
```

Dacă optaţi pentru rezolvarea acestor cerinţe în ClassLang (http://ClassLang.com), completaţi variantele demonstrative următoare. Activarea clasei translator ce procesează limbajul se realizează prin apelul / invocarea acesteia (plasarea numelui pe o linie înaintea codului). Definiţia şi apelul acesteia se pot plasa într-un fişier separat numit Lang, astfel fişierul sursă conţinând doar coduri scrise în limbajul propriu.

```
class MyLang2ClassLang//Interpretor: Interpretare in ClassLang (Front-End)
{
 class line L {L}
 class line a=b {a:=[*b*]}
 class line x=?? {}
 class line Var? .. v {line v}
 class line Output? .. v {#print [/'',v/]}
 class line Input? v {}
}

class MyLang2Cpp{//Translator: Tanslare in C++
 bin 'cpp' \ db '#include <iostream>', 10, 'int main()', 10, '{', 10
 class line L {db [/'',L,';'/], 10}
 class Var1 v=i {db [/'int ',v,=,i,';'/], 10}
 class Var1 v=?? {db [/'int ',v,';'/], 10}
 class line Var? .. V {Var1 V}
 class line Output? .. v {db [/'std::cout<<',v,';'/], 10}
 class line Input? v {db [/'std::cin>>',v,';'/], 10}
 class done {db 'return 0;', 10, '}', 10}
}

class MyLang2Web{//Translator: Tanslare in Server & Client JavaScript + HTML
 bin 'sjs' \ db 'echo(`<html><body><script>', 10//Sau document.write in loc de echo!
 ?:=0
 class line L {db [/'',L/], 10}
 class line Var? .. v {db [/'var ',v/], 10}
 class line Output? .. v {db [/'alert(',v,')'/], 10}
 class line Input? v {db [/'',v,'=Number(prompt("Input ', v, '", "0"))'/], 10}
 class done {db '</script></body></html>`);', 10}
}//Utilizare IntegrativeServer: http://WebTopEnd.com

class MyLang2Win{org 100h//Compilator: Compilare (Bios/Dos/Windows)
 Vars:=
 class line Var? .. v=i {Vars:=Vars \ v db i}//Arhitectura/Stil Harvard
 //class line Var? .. v=i {jmp @f \ v db i \ @@:}//Arhitectura/Stil Princeton
 class line Output? .. v {mov ah, 0eh \ mov al, [v] \ add al, '0' \ int 10h}
 class line Input? v {mov ah, 0 \ int 16h \ sub al, '0' \ mov [v], al}
 class line a=b+c {mov al, [b] \ add al, [c] \ mov [a], al}
 class done {ret Vars}
}

class MyLang2Lin{lin executable//Compilator: Compilare (Unix/Linux/Android)
 Vars:=
 class line Var? .. v=i {Vars:=Vars \ v db i}//Arhitectura/Stil Harvard
 //class line Var? .. v=i {jmp @f \ v db i \ @@:}//Arhitectura/Stil Princeton
 class line Output? .. v {mov al, [v] \ add al, '0' \ call CharToConsole}
 class line Input? v {call KeyFromConsole \ sub al, '0' \ mov [v], al}
 class line a=b+c {mov al, [b] \ add al, [c] \ mov [a], al}
 class done {mov eax, 1 \ mov ebx, 0 \ int 80h
  CharToConsole: push eax \ mov ebx, 1 \ mov eax, 4 \ jmp ConsoleInOut
  KeyFromConsole: push 0 \ mov ebx, 0 \ mov eax, 3
  ConsoleInOut: mov ecx, esp \ mov edx, 1 \ int 80h \ pop eax \ ret
  segment readable writeable Vars
 }
}

class MyLang2Machine{org 100h//Compilator: Compilare/Translare in Coduri/Limbaj Masina
 class line Var? .. v=i {db 0ebh 1 \ v db i}
 class line Output? .. v {db 0b4h, 0eh \ db 0a0h \ dw v \ db 4, '0' \ db 0cdh 10h}
 //class line Output? .. v {db 0a0h \ dw v \ db 24h, 1 \ db 0bah, 0FCh 3 \ db 0eeh}
 class line Input? v {db 0b4h, 0 \ db 0cdh 16h \ db 2ch, '0' \ db 0a2h \ dw v}
 class line a=b+c {db 0a0h \ dw b \ dw 602h, c \ db 0a2h \ dw a}
 class done {db 0c3h}
}
```

Componente şi Descrieri

Orice limbaj poate fi descompus într-o serie de elemente esențial distincte (lexicale, sintactice sau/şi semantice). Gruparea acestor componente în categorii, funcție de rolul lor (funcțional, operativ, informativ, delimitativ, abreviativ etc.), oferă perspectiva unor descrieri compacte a limbajelor prin stabilirea doar a unui set de reguli de combinare a acestor elemente. Astfel, în general, componentele limbajelor aparțin uneia sau mai multora dintre categoriile următoare: caractere speciale, separatori, comentarii, cuvinte rezervate, cuvinte cheie, identificatori, literali, operatori (unari / binari / n-ari, de prefixare / infixare / postfixare / delimitare, aritmetici, relaționali, logici, de selecție, de conversie, condiționali, de atribuire, de secvențiere), expresii, instrucțiuni (de atribuire, de intrare/ieşire, de control condiționat, de ciclare, de transfer/salt, de subprogramare, de compilare condiționată, de includere), metode, fraze/blocuri multi-instrucțiune.

Formatele de descriere ale limbajelor reprezintă modalități de caracterizare formal-sintetică a acestora plecând de la componentele limbajelor, prin utilizarea unor reguli specifice de combinare a elementelor acestora. Un format de descriere trebuie să fie concis (simplu), precis (fără ambiguități), formal (cu reguli parsabile şi să poată fi implementat pe sisteme de calcul), natural (cu notații sugestive), general (capabil de a descrie limbaje complexe), auto-descriptiv (să se descrie complet pe sine), liniar (să utilizeze expresii formate din şiruri de caractere). Limbajele de descriere (geometrizare) a unor informații, limbaje sau procese (transformări) se mai numesc şi metalimbaje sintactice.

Formatul **BNF** (**B**ackus–**N**aur **F**orm/**B**ackus **N**ormal **F**orm) de descriere a limbajelor este compus din producții (reguli), specificate prin intermediul unor meta-simboluri: `<..>` marcarea producțiilor (delimitare nume), `::=` operatorul de definire a unei producții, `|` separator al definițiilor alternative, `".."` delimitare simboluri lexicale sau terminale. Exemple de descrieri în formatul BNF:

```
<Cifra>   ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<Natural> ::= <Cifra> | <Cifra> <Natural>
<Semn>    ::= "+" | "-"
<Intreg>  ::= <Semn> <Natural> | <Natural>
<Litera>  ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
              "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
              "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" |
              "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" |
              "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
              "y" | "z"
<Cuvant>  ::= <Litera> | <Litera> <Cuvant>
```

Formatul **WSN** (**W**irth **S**yntax **N**otation) de descriere a limbajelor este compus din producții, marcate de: `=` operatorul de definire a unei producții, `.` simbol de marcare a sfârşitului producției, `|` separator al definițiilor alternative, `[..]` delimitare definiții opționale, `{..}` delimitare definiții repetitive, `(..)` delimitare definiții grupate, `".."` sau `'..'` delimitare stringuri terminale. Formatul permite utilizarea unor descrieri iterative, comparativ cu BNF care utilizează doar reguli recursive de compunere. Parte din notații vor fi incluse în EBNF.

Formatul **EBNF** (**E**xtended **B**ackus–**N**aur **F**orm/**E**xtended **B**ackus **N**ormal **F**orm) este compus din producții marcate prin: `=` operatorul de definire a unei producții, `.` sau `;` marcarea sfârşitului producției, `|` separator al definițiilor alternative, `[..]` delimitare definiții opționale, `{..}` delimitare repetiții sau N`*`E pentru multiplicare, `(..)` delimitare definiții grupate, `".."` sau `'..'` delimitare stringuri terminale, `,`

operator de concatenare, `(*..*)` comentariu, `?..?` secvență specială (extensie), `-` excepție (excludere). Exemple de descrieri în formatul EBNF:

```
Cifra    = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
Natural = Cifra , {Cifra} ;
Semn     = "+" | "-" ;
Intreg   = [Semn] Natural ;
Paronime1  = "familia" ("l" | "r") ;
(* Echivalent cu *)
Paronime2  = "familial" | "familiar" ;
```

Formatul **ABNF** (**A**ugmented **B**ackus–**N**aur **F**orm) utilizează producții (reguli) case-insensitive pentru neterminali, fiind definit prin: `<..>` delimitare opțională pentru nume reguli, `=` definiție, `;` comentariu linie, spațiu sau `.` pentru concatenare, `/` separator reguli alternative, `=/` adăugarea de noi reguli alternative, N`*`R multiplicare reguli, `(..)` delimitare reguli grupate, `[..]` delimitare reguli opționale, `WSP` spațiu liniar, `LWSP` spațiu linii, `-` operator de infixare interval, `%B..` valoare numerică în baza/format `B` (b = binar, d = zecimal, x = hexa), `%s` prefixare pentru case-sensitive, `%i` prefixare pentru case-insensitive, `".."` sau `'..'` delimitare stringuri terminale. Exemple de descrieri în formatul ABNF:

```
Digit    = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
Cifra    = %x30-39 ; Ca valoare din interval, Echivalent cu Digit
Natural = 1*Cifra
Fruct    = "Mar" / "Pruna" / "Strugure"
Fruct    =/ "Portocala"
CR       = %x0D ; Carriage Return
LF       = %x0A ; Line Feed
CRLF     = CR LF
CRLF     = %x0D.0A ; Varianta CRLF
```

1.● Descrieți limbajul MyLang implementat anterior, în cel puțin unul din formatele prezentate

2. ○ Descrieți într-unul din formatele prezentate, elemente ale limbajelor umane/naturale (En şi/sau Ro) cum ar fi formarea/conjugarea unor clase/grupe de verbe regulate sau/şi neregulate, declinări etc. sau informații/sisteme cu structură/geometrie fractală (http://ClassLang.com/?Refs=Languages/Fractal)

3. ● Descrieți în cel puțin unul din formatele prezentate, limbajul pe care îl cunoaşteți cel mai bine (sau componente relevante din acesta).

4. ○ Descrieți în cel puțin unul din formatele prezentate, modul de formare a identificatorilor unor resurse oarecare (URI - Uniform Resource Identifier) şi a identificatorilor/adreselor resurselor Web (URL - Uniform Resource Locator). Identificați componentele URI/URL aferente unei aplicații Web completând implementările următoare (IntegrativeServer from http://WebTopEnd.com)

```
var URL=RequestProtocol + "://" + RequestHost + RequestPort + RequestPath
+ "?" + RequestQuery; echo(URL);//sjs: Server JavaScript
```

```
Dim URL: URL=RequestProtocol & "://" & RequestHost & RequestPort & _
RequestPath & "?" & RequestQuery: Echo(URL)'svb:Server Visual Basic script
```

5. ○ Implementați un (meta)program care să primească ca inputuri un fişier cu reguli descrise în unul din formatele prezentate anterior (câteva reguli/producții) şi unul cu un cod scris în limbajul descris de acele reguli, programul urmând să efectueze analiza (parsarea/validarea) sintactico-lexicală a codului.

Descrieri Informale

Descrierile informale (incomplet formale) se bazează pe reguli (gramaticale) flexibile care permit libertăți lingvistice sau artistice în structurarea unor informații prin intermediul unor (pseudo) limbaje sau formate de marcare/reprezentare/documentare/codare care structurează conținut cu adresabilitate finală fie umană, fie umană şi sistem de calcul. Limbajul **TeX/LaTeX** permite descrierea / reprezentarea grafică a informației cu orice grad de complexitate sau structurare (http://tug.org) precum cărți, lucrări, articole etc. Limbajul **XML** (e**X**tensible **M**arkup **L**anguage) descrie portabil informații/date (în vederea stocării sau transportului) prin intermediul unor de tag-uri/repere/etichete cărora li se pot ataşa eventual diferite atribute (http://www.w3.org/XML, https://www.w3schools.com/xml). Limbajul **HTML** (**H**yper**T**ext **M**arkup **L**anguage) are o sintaxa similară cu **XML** dar cu instrucțiuni/tag-uri/etichete prestabilite, case-insensitive, cu scopul de descriere / reprezentare grafică a unor informații (https://www.w3.org/html, https://www.w3schools.com/html). Formatul **JSON** (**J**ava**S**cript **O**bject **N**otation) permite descrierea simplificată/compactă a datelor prin asocieri de tipul nume:valoare (http://www.json.org, https://www.w3schools.com/js). Alte formate document relevante: **DOC(X)**, **PS/PDF**, **RTF**, **MD, TXT**.

1. Descrieți în cel puțin două formate (XML şi HTML de exemplu) structura/componentele tezei de
- licență, la modul general după structura tezei specificată pe site-ul facultății şi adaptată temei proprii. Se vor urmări rigoarea elementelor prezentate (topica, ierarhia, conținut minimal relevant – câteva referințe bibliografice, câte o frază la nivelul fiecărui (sub)capitol etc.) şi descrierea corectă în limba română (cu diacritice etc.). Componentele generale ale unei lucrări de licență: I. Introducere (istoric temă, abordări similare cu citări, avantaje, limite etc.), II. Fundamentarea teoretică şi tehnologiile utilizate (problematica domeniului şi elemente generale, teoretice şi tehnologice ale domeniul temei, clasificări, metode etc. - maxim 30% din lucrare), III. Componentele şi caracteristicile metodei sau/şi a aplicației propuse (descrierea problemei, modurilor de soluționare/implementare propuse etc.), IV. Implementarea, gestiunea, testarea şi utilizarea metodei/aplicației (detalii cu privire la modul concret în care a fost implementată, testată, verificată aplicația, metoda sau tehnologia propusă, despre cum ar trebui un utilizator sa o folosească, despre utilitate, limite, perspective etc.) V. Concluzii (se prezentă succint în cate o frază elementele esențiale ale soluției propuse), VI. Bibliografie (listă cu cărțile, articolele sau resursele web utilizate), VII. Anexe (detalii ale elementelor teoretice, coduri mai ample ale implementărilor etc.). Permiteți accesarea informațiilor prin intermediul unui server, eventual restrictiv unor colegi/stații (`if(RequestIP==IPok)echo(ReadFileContent('..'))`).

Formatul **pseudocod** reprezentă modalitatea de descriere simbolică, sugestivă, (in)formal textuală sau matematică a unor date, operații, coduri, algoritmi, procese etc. Descrierea se poate realiza în orice limbă şi prin similarități cu orice limbaj, permițând focusarea pe esența unui cod sau algoritm, într-o manieră simplă şi concisă, fără a fi distras de sintaxa unui limbaj de programare. Există o multitudine de dialecte pseudocod, după stilul descriptiv al fiecăruia, cele mai utilizate dialecte fiind: General (Ro, En etc.) – instrucțiuni sugestive şi sintaxa similară Basic, Matematic – descrieri similare stilului matematic, C, Pascal etc. Pseudocodurile se pot interpreta destul de uşor direct, se pot transla în instrucțiuni echivalente ale limbajelor de programare, sau chiar se pot compila (transla în instrucțiuni procesor/maşină), aşa cum se poate observa din analiza/testarea implementărilor ClassLang (http://ClassLang.com) următoare.

```
class PseudoCodeInterpreter//Interpretor: Interpretare (Back-End)
{
 class line L {L}
 class line VAR? .. V {V}
 class line PRINT? V{@print V}
 class line WHILE? C {@while C}
 class line ENDW? {@endw}
}
```

```
class PseudoCode2Py{//Translator: Tanslare in Python
 bin 'py' \ Indent=0 \ I_=9//Sau 32(' ')
 class line L {rb Indent, I_ \ db ['',L/], 10}
 class line VAR? .. V{rb Indent, I_ \ db ['',V/], 10}
 class line PRINT? V{rb Indent, I_ \ db ['print ',V/], 10}
 class line WHILE? C{rb Indent, I_\db ['while ',C, ':'/],10\Indent=Indent+1}
 class line ENDW? {Indent=Indent-1}
}

class PseudoCodeCompiler{//Compilator: Compilare (Bios-Dos-Windows)
 dos \ push D \ pop ds
 Vars:=
 class line VAR? .. v=i {Vars:=Vars \ v db i}
 class line WHILE? a<i {startW: cmp [a], i \ jge endW}
 class line ENDW? {jmp startW \ endW:}
 class line PRINT? v{mov ah, 0eh \ mov al,[v] \ int 10h}
 class line a=b+i {mov al, [b] \ add al, i \ mov [a], al}
 class done {mov ah, 0 \ int 16h \ mov ax, 4C00h \ int 21h \ segment D Vars}
}

class PseudoCode2Machine{org 100h//Compilare/Translare in Limbaj Masina
 class line VAR? .. v=i{db 0ebh @f-$-1 \ v db i \ @@:}
 class line WHILE? a<i {startW: dw 3e80h a \ db i \ db 7dh endW-$-1}
 class line ENDW? {db 0ebh startW-$-1 \ endW:}
 class line PRINT? v{db 0b4h, 0eh \ db 0a0h \ dw v \ db 0cdh 10h}
 //class line PRINT? v{db 0a0h \ dw v \ db 24h, 1 \ db 0bah,0FCh 3 \ db 0eeh}
 class line a=b+i {db 0a0h \ dw b \ db 4, i \ db 0a2h \ dw a}
 class done {db 0b4h, 0 \ db 0cdh 16h \ db 0c3h}
}

class PseudoCode2SJS{//Translator: Tanslare in Server JavaScript
 bin 'sjs'//'js' pentru Client JavaScript
 class line L {db [/'',L/], 10}
 class line VAR? .. V{db [/'var ',V,';'/], 10}
 class line PRINT? V{db [/'document.write(',V,');'/], 10}//sau echo
 class line WHILE? C{db [/'while(',C, '){'/], 10}
 class line ENDW? {db '}', 10}
}//Utilizare IntegrativeServer: http://WebTopEnd.com

class PseudoCode2SVB{//Translator: Tanslare in Server Visual Basic Script
 bin 'svb'//'vbs' pentru Client Visual Basic Script
 class line L {db [/'',L/], 10}
 class line VAR? .. V=I{db [/'Dim ',V,':', V=I/], 10}
 class line PRINT? V{db [/'Document.Write(',V,')'/], 10}//sau Echo
 class line WHILE? C{db [/'Do While ',C/], 10}
 class line ENDW? {db 'Loop', 10}
}

PseudoCodeInterpreter//Invocare/Apel/Activare. Exemplu Pseudocod:

VAR C = 0
WHILE C < 127
 PRINT C
 C = C + 1
ENDW
```

2. Implementați un program care să realizeze interpretarea, translatarea într-un alt limbaj sau compilarea
   - unor instrucțiuni pseudocod.
3. Implementați un translator care să convertească coduri scrise într-un limbaj oarecare în
   - corespondentul pseudocod al acestora.

Simboluri

Fie că descriem anumite informaţii într-un limbaj fie că un limbaj utilizează anumiţi operatori sau caractere, procesarea informaţiilor se reduce în ultimă instanţă la diferite procese de translare şi codare a unor simboluri (ce ar putea reprezenta de exemplu simbolul ∓?). Sistemele de calcul nu operează direct cu simboluri grafice ci cu cantităţi numerice asociate (de preferinţă bijectiv) acestora. Codarea simbolurilor presupune în principal a indexa tipul simbolului şi în secundar a descrie / identifica anumite caracteristici ale acestuia (elemente de stil precum designul simbolului, înclinare, îngroşare etc.). Codarea se poate realiza pe un număr fix sau variabil de octeţi asociaţi simbolurilor. Codarea pe număr constant de octeţi are avantajul procesării rapide (datorita indexării) dar, dacă se realizează pe mai mulţi octeţi, poate genera dificultăţi în cazul protocoalelor de comunicaţii, în special celor structurate pe schimburi de octeţi. Codarea pe număr variabil de octeţi are avantajul compatibilităţii maxime cu implementările anterioare, dar şi dezavantajul procesării mai lente din cauza lipsei indexării directe. Vizualizare/Editare simboluri sau conţinut binar http://classlang.com/?Refs=None/Noname0&Edit=Binary&File=Aa%0D%0AZz%20, reprezentări/translări ale valorilor/simbolurilor http://classlang.com/?Refs=None/Noname0&Search=350

Unicode (http://www.unicode.org) este standardul pentru indexarea digitală a simbolurilor precum: caractere ale limbilor globului, scripturi antice, simboluri matematice, sigle etc. Simbolurile sunt grupate pe diferite categorii: Multilingual Basic (0000–FFFF) & Supplementary (10000–1FFFF), Ideographic (20000–2FFFF) etc. Referirea la un simbol Unicode se specifică prin: **U+IndexHexa** (Exemplu **A=U+41**), utilizarea în diferite limbaje a unui simbol necesitând specificarea indexului acestuia (sub forma zecimală sau hexa), de exemplu **&#d;**, **&#xh;**, **\u..** etc.

Cele mai reprezentative standarde de codare pe număr constant de octeţi sunt: **ASCII/US-ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), **ISO/IEC-8859-1..ISO/IEC-8859-16** (16 categorii ce codifică simboluri pentru diferite grupuri de limbi)**, UCS-2** (**2**-byte **U**niversal **C**haracter **S**et, codare pe 16 biţi a simbolurilor Unicode în intervalul **U+0000 .. U+FFFF**), **UTF-32 / UCS-4** (**U**nicode **T**ransformation **F**ormat in **32** bits/**4**-byte **U**niversal **C**haracter **S**et).

În cazul codărilor multi-octet, ordinea plasări octeţilor poate fi de tip **Little-Endian** (octetul nesemnificativ primul), **Big-Endian** (octetul semnificativ primul) sau **Middle/Mixed-Endian** (combinaţii ale celor două). Exemplu: $1100001011000100110001101100100_2$ = 14130461544_8 = 1633837924_{10} = 61626364_{16} = 64636261_{16LE} = $abcd_{BE}$ = $dcba_{LE}$ = $badc_{MLE}$. Translări din/în orice bază: http://classlang.com/?Translate=abcd&FromBase=16&ToBase=10LE

Cele mai reprezentative standarde de codare pe un număr variabil de octeţi sunt: **UTF-8** (**U**nicode **T**ransformation **F**ormat in **8** bits) şi **UTF-16** (**U**nicode **T**ransformation **F**ormat in **16** bits).

UTF-8 este o codare flexibilă a simbolurilor pe un număr variabil de la 1 până la 4 octeţi realizată prin intermediul transformărilor:

$U+0000..U+007F \Rightarrow 0xxxxxxx_2$

$U+0080..U+07FF \Rightarrow 110xxxxx_2\ 10xxxxxx_2$

$U+0800..U+FFFF \Rightarrow 1110xxxx_2\ 10xxxxxx_2\ 10xxxxxx_2$

$U+10000..U+10FFFF \Rightarrow 11110xxx_2\ 10xxxxxx_2\ 10xxxxxx_2\ 10xxxxxx_2$

Exemple:

S=S=S=1010011_2, U+053 ⇒ 01010011_2 = 53_{16}

Ş=Ş=Ş=101011110_2, U+015E ⇒ 11000101_2 10011110_2 = $C59E_{16}$

𝚹=𝚹=𝚹 = 11101011010111001_2, U+1D6B9 ⇒ 11110000_2 10011101_2 10011010_2 10111001_2 = $F09D9AB9_{16}$

http://classlang.com/?Refs=None/Noname0&Edit=Binary&File=%53%C5%9E%F0%9D%9A%B9

UTF-16, cu variantele sale **UTF-16BE/UTF-16LE** – **B**ig/**L**ittle-**E**ndian, utilizează 2 sau 4 octeţi pentru codare prin intermediul următoarelor reguli de transformare:

U+D800..U+DFFF interval folosit pentru codări extinse
U+10000..U+10FFFF = U ⇒ $HS_{(16)}$ $LS_{(16)}$
$U_{(20)}$ = U - 10000_{16} valoare pe 20 de biți
$HS_{(16)}$ (High/First Surrogate) = $D800_{16}$ + ($U_{(20)}$ >> 10)
$LS_{(16)}$ (Low/Second Surrogate) = $DC00_{16}$ + ($U_{(20)}$ & $3FF_{16}$)

Exemple:

S=S=S=1010011_2, U+053 ⇒ 0053_{16}

Ş=Ş=Ş=101011110_2, U+015E ⇒ $015E_{16}$

𝚹=𝚹=𝚹 U+1D6B9 ⇒ $U_{(20)}$=$1D6B9_{16}$-10000_{16}=$D6B9_{16}$=
1101011010111001_2 ⇒ HS=$D835_{16}$, LS=$DEB9_{16}$

http://classlang.com/?Refs=None/Noname0&Edit=Binary&File=%00%53%01%5E%D8%35%DE%B9

http://classlang.com/?Refs=None/Noname0&Edit=Binary&File=%53%00%5E%01%35%D8%B9%DE

1. ● Implementați metode proprii de conversie a unui text lowercase scris în limba engleză şi/sau română (cu diacritice etc.) în varianta uppercase şi/sau invers şi/sau metode case-insensitive de căutare a unui substring într-un text.

2. ○ Explorați la nivel de octet conținutul unui fişier şi afişați indexul Ascii şi Unicode al fiecărui simbol identificat

3. ● Implementați metode de conversie a unor octeți (conținutul unui fişier de exemplu) ce codează simboluri Unicode (cel puțin câte 1 simbol din fiecare dintre cele 4 intervale specifice UTF-8) din UTF-8 în coduri UTF-16/32 şi/sau metoda de conversie inversă. Metodele, împreună cu alte operații specifice (citire/scriere, indexare, concatenare etc.) putând fi încapsulate într-un obiect String.

4. ○ Implementați o metodă de contorizare a numărului de simboluri/caractere dintr-un (fişier) text scris în limba română şi codat în UTF-8/16 sau un translator direct şi invers a unor texte scrise cu diacritice în formatul fără diacritice sau a unor texte scrise în alfabetul chirilic respectiv latin.

5. ○ Reprezentați grupările serie şi paralel ale unor capacitoare simbolizate prin ⫟.

Literali

Literalii reprezintă componente ale limbajelor sau translatoarelor compuse din grupuri de simboluri care au valori constante, utilizate în expresii pentru iniţializarea/popularea variabilelor, abrevieri, definiţii, denumiri, (ne)terminali, indexări etc. Un literal este compus dintr-o serie de simboluri eventual prefixate, postfixate, infixate sau/şi delimitate de nişte simboluri reper. Exemple de literali: **-101, 101b, 1.5E+6, 0xF6, 0F6h, $F6, "AB\t1", 'Q\u501', #FF00FF, ŧ, ɀ, {5, 8}, ABC**. În sens larg, entităţile literale reprezintă combinaţii compacte de simboluri cu semnificaţii sintactico-lexicale (atomi sintactici), incluzând cuvintele şi semnele de punctuaţie dintr-o limbă etc.

În procesul compilării sau translatării codurilor, literalii trebuie convertiţi în reprezentarea internă a limbajului sau a sistemului de calcul (binară, BCD, Gray etc.), în valori pe un şir de biţi sau pe unul sau mai mulţi octeţi, uneori fiind necesară şi operaţia inversă. Literalii conţin şiruri sau grupuri de şiruri de simboluri ce reprezintă valori numerice descrise în diferite baze, denumiri etc. Cele mai utilizate baze de descriere a valorilor numerice sunt prezentate în cele ce urmează.

01_2=01b, 1011_2=1011_{2BE}=1101_{2LE}=1011b=1110g (g: Gray/Reflected Binary Code – RBC)
01234567_8=0o01234567=01234567o=01234567=\01234567
0123456789_{10}=0123456789=0123456789d=&#0123456789;
$0123456789ABCDEF_{16}$=0x0123456789ABCDEF=$0123456789ABCDEF=0123456789ABCDEFh=
=\x0123456789ABCDEF=&#x0123456789ABCDEF;
$0123456789abcdef_{16}$=0x0123456789abcdef=$0123456789abcdef=0123456789abcdefh=
=\x0123456789abcdef=&#x0123456789abcdef;
$ABCDEFGHIJKLMNOPQRSTUVWXYZ234567_{32}$=ABCDEFGHIJKLMNOPQRSTUVWXYZ234567
$123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz_{58}$=123456789ABCDEFGH
JKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
+/0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz$_{64}$=+/012345678
9ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
S┼₨Γåé≡¥ôÉ$_{256\,(UTF-8)}$=53C59EE28682F09D9390h$_{(UTF-8)}$="SŞⒹ𝒜"
S ^☺é!5╪╨▄$_{256\,(UTF-16)}$=53005E01822135D8D0DCh$_{(UTF-16)}$="SŞⒹ𝒜"
S ^☺ é! ╨└☺$_{256\,(UTF-32)}$=530000005E01000082210000D0D40100h$_{(UTF-32)}$="SŞⒹ𝒜"
$ABC123_{256}$=ABC123$_{256BE}$=321CBA$_{256LE}$=ABC123$_{(256)}$=ABC123$_{(U)}$="ABC123"

Translări din/în orice bază: http://classlang.com/?Translate=10&FromBase=10BE&ToBase=2LE

După identificarea şi delimitarea şirurilor de caractere/simboluri asociate valorii codate (fără simbolurile delimitatoare/bazei precum prefixele 0x, $, " sau/şi sufixele b, h, " etc.) literalii trebuie convertiţi/translaţi în reprezentări binare (interne) cu care limbajul sau sistemul de calcul destinaţie operează. Conversia unor secvenţe textuale (literali) în conţinut binar reprezintă cea mai utilizată operaţie de translare a codurilor sau a procesării codurilor scrise în diferite limbaje. Procesarea literalilor numerici se reduce la utilizarea repetată directă şi/sau eventual inversă a algoritmului de conversie a unui întreg din reprezentarea literală **S** (general simbolică/matematică de tip Big-Endian) în baza **B** în forma binară (internă) **I**.

```
FUNCTION StrToInt(S, nS, B)
  I=0
  FOR J=0 TO nS-1 DO
    I = I * B + SymbolCodeToInt(S[J], B)
  RETURN I
```

Există operaţii în cadrul limbajelor sau al translatoarelor pentru care este nevoie să realizăm conversii inverse ale unor numere din conţinutul binar în format text/literal (preprocesări, specificarea numerică a

unor caractere în stringuri, de exemplu \u.. , translări în limbaje ce suportă alte baze de reprezentare etc.). Algoritmul de conversie a unui întreg **I** în reprezentarea literală **S** în baza **B** este următorul

```
FUNCTION IntToStr(I, B, S)
  nS = 0
  REPEAT
    S[nS++] = IntToSymbolCode(I%B, B)
    I = I / B
  UNTIL I==0
  RETURN nS
```

Literalul numeric astfel obținut este în reprezentarea Little-Endian, reprezentarea uzuală general simbolică/matematică de tip Big-Endian obținându-se prin inversarea succesiunii simbolurilor acestuia.

```
PROCEDURE Reverse(S, nS)//Little-Endian ⇔ Big-Endian
  L = 0
  R = nS-1
  WHILE L < R DO
    S[L++] ⇔ S[R--]
```

Unele limbaje permit construirea unor entități de tip string prin intermediul unor şabloane/paternuri/template-uri literale (template literals/strings) ca variante mai lizibile ale operațiilor echivalente de translare şi concatenare. Exemple: `"\t%d\r\n"`, `"V=$V"`, `` `V+1=${V+1}` `` etc.

1.● Implementați o metodă de identificare (şi afişare eventual) a tuturor entităților literale dintr-un text

2. ○ Identificați şi translați literalii maşină (opcod) următori atât în cod executabil cât şi în echivalentul acestora în limbaje de nivel mai înalt: `\264\11\272\14\1\315\41\264\0\315\26\303Hi!$` în reprezentarea octală şi ASCII sau `10110100b 1001b 10111010b 1100b 1b 11001101b 100001b 10110100b 0b 11001101b 10110b 11000011b 1001000b 1101001b 100001b 100100b` în reprezentarea binară sau `0B4h 9h 0BAh 0Ch 1h 0CDh 21h 0B4h 0h 0CDh 16h 0C3h 48h 69h 21h 24h` în reprezentarea hexazecimală sau ┤○‖♀☺═!┤ ═╼├Hi!$ în baza 256.

3. ● Implementați o metodă de schimbare a bazei de reprezentare a unui număr întreg, scris textual într-un string sau într-un fişier text. Generalizați metoda pentru un număr indefinit de mare şi implementați metode de conversie a unui număr real dintr-o formă text/simbolică în forma numerică internă specifică limbajului/sistemului utilizat.

4. ○ Implementați metode de translare directă şi inversă a unor literali numerici din orice bază în literali/coduri Gray (**RBC** – **R**eflected **B**inary **C**ode).

5. ○ Procesați literali formați cu simboluri indexate pe poziții Unicode superioare, de exemplu valori numerice descrise cu simbolurile **U+FF10..U+FF19** = ０ １ ２ ３ ４ ５ ６ ７ ８ ９, **U+2460..U+2468** = ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ etc.

6. ○ Implementați metode de translare/procesare a unor şabloane/paternuri/template-uri literale (template literals/strings) cu caracteristici similare cu cele utilizate în diferite limbaje precum JavaScript (JS/SJS), ServerVisualBasic (SVB), PHP, C# etc. Exemple de concatenări string şi şabloane literale:

```
echo(RequestProtocol+"://"+RequestHost+RequestPath)//Strings Concatenation
echo(RequestProtocol & "://" & RequestHost & RequestPath)//Concatenation
echo(`${RequestProtocol}://${RequestHost+RequestPath}`)//Template Literals
echo "http://".$_SERVER['HTTP_HOST'].$_SERVER['REQUEST_URI'];//Strings
echo "http://{$_SERVER['HTTP_HOST']}{$_SERVER['REQUEST_URI']}";//Literals
```

Expresii

Expresiile reprezintă combinații de literali (operatori/operanzi: cuvinte, variabile, constante, separatori) ce descriu informații corelate sau procese de calcul care prin evaluare exprimă cantități echivalente acestora. Procesul de evaluare poate să producă si efecte secundare (side effects) dacă se modifică şi parametrii externi expresiei (precum în `C+=(A++)+B|9`). Procesarea expresiilor presupune identificarea operațiilor (de prefixare, postfixare sau infixare) şi pe baza precedențelor / priorităților acestora şi a tipului de asociativitate se gestionează logic elementele expresiei prin combinarea cel mult a două cantități. Funcție de aritatea operațiilor, expresiile pot conține operații unare, binare, trinare etc. Funcție de tipul datelor asupra cărora operează, expresiile pot fi aritmetice, logice, ordinale etc. Funcție de elementele asupra cărora operează, expresiile pot produce modificări ale unor variabile (incrementare, atribuire, multiplicare etc.) sau pot valida sau extrage anumite componente (expresii regulate).

1. Descrieți în BNF, WSN, EBNF sau ABNF formatul/formarea expresiilor în limbajul pe care îl
   - cunoaşteți cel mai bine.

2. Implementați un translator Român – Englez direct şi/sau invers a unor expresii plecând de la lexic şi
   - prin procesarea corelațiilor substantiv – adjectiv – verb ca operatori-operanzi pre/post fixați corespunzător fiecărei limbi (Ex: cer albastru strălucitor ⇔ bright blue sky) etc.

3. Implementați o metodă de procesare a unor expresii specifice unui limbaj prin translarea expresiei în
   - valoarea echivalentă (evaluator/interpretor/translator/compilator), în corespondentul expresiei în alt limbaj sau în o expresie echivalentă prin modificarea operatorilor din forma infix în forma postfix, prefix sau funcțională etc. Dacă optați pentru rezolvarea acestor cerințe în ClassLang (http://ClassLang.com) se pot utiliza/completa implementările următoare.

```
//Parser/Evluator de Expresii si Operatii dupa prioritati si asocieri.
class E a {E.res:=a}//Atom sintactic (literal)
class E (a) {E a}//Gestiune/Eliberare paranteza (prioritate impusa)
class E a*b {E.res:=Mul(E(a),E(b))}//Asociere oarecare (implicit dreapta)
class E a+b {E.res:=Sum(E(a),E(b))}//Asociere oarecare (implicit dreapta)
class E a-b {E.res:=Sub(E(a),E(b))}//Asociere stanga (see next class)
class E a-b-c {E (a-b)-c}//Necesitate/Impunere/Constructie asociere stanga
class E a==b {E.res:=Eq(E(a),E(b))}//Testare egalitate


class Interpreter
{
 bin 'txt'
 class Sum a,b {Sum.res:=[*a+b*]}
 class Eq a,b  {Eq.res:=a==b}
}
Interpreter
db [/'' E(4+5)/]

class Translator
{
 bin 'pas'
 class Sum a,b {Sum.res:=a+b}
 class Eq a,b  {Eq.res:=a=b}
}
//Translator
//db [/'' E(X==Y)/]
```

```
class Compiler
{
 class Sum a,b {mov al,a \ add al, b} \\ Sum.res:=al
 class Eq a,b {mov al, a \ cmp al, b \ sete al} \\ Eq.res:=al
}
//Compiler
//X = 1 \ Y = 2 \ Z = 3//X db 1 \ Y db 2 \ Z db 3
//E X+Y==Z
```

Expresiile regulate (**RegEx**/**RegExp** - **Reg**ular **Exp**ression) reprezintă paternuri/şabloane formate din succesiuni de simboluri şi reguli (meta-caractere) ce identifică, extrag sau înlocuiesc secvenţe de text. Expresiile regulate sunt definite prin: **/../** sau **".."** delimitare reguli, **.** orice caracter (mai puţin sfârşitul de linie) sau un literal în **[.]**, **^..** început şir/linie, **..$** sfârşit şir/linie, **\c** caracter **c** special/excepţie, **(..)** grupare, **|** separator opţiuni/alternative, **[..]** set simboluri posibile, **[^..]** set simboluri excluse, **..-..** infixare interval, **..+** minim o repetare, **..?** maxim o repetare, **..*** oricâte repetări, **..{m, M}** repetări între **m** şi **M**, **<..>** cuvânt, **\d** digit, **\D** non-digit, **\s** spaţiu, **\S** non-spaţiu, **\w** alfanumeric, **\W** non-alfanumeric, **\b** bordul/marginea cuvântului, **\A** început de şir, **\z** sfârşit de şir. Exemple de expresii regulate:

```
(a|b)* ⇒ "", "a", "b", "aa", "ab", "ba", ..
[abc0-2] ⇔ [a-c0-2] ⇒ a, b, c, 0, 1, 2
A{1,5} ⇒ "A", "AA", "AAA", "AAAA", "AAAAA"
[0-7]+o ⇒ 0o, 1o, 7o, 01o, 57o, ..
[0-9]+[dD]? ⇒ 0, 1, 0d, 35, 490D, ..
\w ⇔ [a-zA-Z0-9_] ⇒ A, 1, _, z, ..
\W ⇔ [^a-zA-Z0-9_] ⇒ +, !, * / \ % $ @ ..
\b ⇔ (^\w|\w$|\W\w|\w\W)
[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z0-9]+ ⇒ hocus@pocus.com, ..
\w+@\w+\.\w+ ⇒ ala@bala.portocala, hocus@pocus.com, ..
```

Standardizarea **POSIX** (Portable Operating System Interface) a expresiilor regulate cuprinde **SRE**/**BRE**/**ERE** (**S**imple/**B**asic/**E**xtended **R**egular **E**xpressions) în care unele dintre regulile-simbol de bază sunt completate/înlocuite cu: **\p{ASCII}** ASCII, **[:alnum:]** / **\p{Alnum}** alfanumerice, **[:word:]** alfanumerice şi **_**, **[:alpha:]** / **\p{Alpha}** / **\a** alfabetice, **[:blank:]** / **\p{Blank}** / **[:space:]** spaţiu, **\S** non-spaţiu, **\B** non-bord, **[:cntrl:]** / **\p{Cntrl}** control, **[:digit:]** digit, **[:graph:]** / **\p{Graph}** vizibile, **[:print:]** / **\p{Print}** printabile, **[:punct:]** / **\p{Punct}** punctuaţie, **[:lower:]** lowercase, **[:upper:]** uppercase, **[:xdigit:]** hexa.

4. ● Implementaţi metode de procesare a unor expresii regulate cu câteva reguli (de exemplu pentru extragerea adreselor de email dintr-un text) şi comparaţi rezultatul cu utilizarea aceloraşi reguli în cadrul unui limbaj ce are implementate metode de procesare a expresiilor regulate.

5. ○ Implementaţi metode de combinare a ***expresiilor regulate*** cu ***şabloane literale***. De exemplu utilizarea metodelor RegExp JavaScript la nivelul client pentru a obţine componentele URL-ului curent (identificate eventual precum în cazul server în variabile denumite **`RequestProtocol`**, **`RequestHost`**, **`RequestPath`**, **`RequestQuery`** etc.) şi utilizarea acestora în construirea unor link-uri (referirea altor resurse web de pe server) prin intermediul unor şabloane literale.

Fraze

Frazele reprezintă grupuri de propoziţii între care există corelaţii (de coordonare sau subordonare) şi care pot fi privite sintactic şi semantic unitar/autonom. Propoziţiile reprezintă cele mai mici entităţi sintactice (semi)autonome adică reprezintă combinaţii lexicale cu anumite semnificaţii sau corelări/grupări ale unor expresii. În cazul limbajelor de programare propoziţiilor sau frazelor le corespund instrucţiunile simple sau complexe (compuse/corelate), formate din combinaţii de expresii evaluabile de un număr de ori, condiţionat unele funcţie de altele. Instrucţiunile pot fi de diferite tipuri: preprocesare (procesare text), procesare/codare, atribuire, intrare/ieşire, control (transfer/salt, condiţionale, ciclare) etc. Blocurile multi-instrucţiune reprezintă un grup de propoziţii/instrucţiuni între care există corelaţii (de coordonare, subordonare sau incluziune). Implementarea instrucţiunilor presupune utilizarea unor repere de execuţie sau de salt (etichete sau label-uri) prin intermediul cărora se execută selectiv/condiţionat anumite expresii sau (sub)instrucţiuni (statements). De exemplu, instrucţiunea `if Condition then StatementIf else StatementElse endif` se implementează şi se interpretează/execută de către sistemele de calcul conform descrierii formale:

```
NEW LabelElse, LabelEndIf
IF !Condition THEN GOTO LabelElse
  StatementIf
  GOTO LabelEndIf
LabelElse:
  StatementElse
LabelEndIf:
OLD LabelEndIf, LabelElse
```

În mod similar se descriu şi se implementează majoritatea instrucţiunilor (`switch-case`, `while-do`, `do-while`, `repeat-until`, `for-next` etc.).

1. Pentru a sublinia modul de procesare specific sistemelor de calcul şi în acelaşi timp a demistifica
● anatemizarea (adesea din ignoranţă) instrucţiunii **`GoTo`** (totuşi cea mai simplă şi utilizată instrucţiune internă/low-level/maşină de control/salt/transfer necondiţionat, utilizată pentru implementarea tuturor instrucţiunilor de control/ciclare), a se implementa un algoritm simplu (de sortare de exemplu) în care să se utilizeze într-un limbaj mid-level or high-level (C/C++ de exemplu) doar instrucţiunile **`GoTo`** şi **`If`** (sau instrucţiuni **`jmp/cmp`** procesor) în loc de **`For`** sau **`While`**, adică de tip "Flat(tened) C".

2.○ Implementaţi un translator direct şi invers al unor fraze din limba română în limba engleză.

3. Implementaţi un interpretor sau un translator al unui cod (descris într-un string sau un fişier text) care
● să conţină cel puţin două instrucţiuni corelate/întrepătrunse/imbricate.

4. Similar cu implementarea demonstrativă a instrucţiunii **`IF-ELSE`** următoare, să se implementeze şi
○ să se testeze în ClassLang (http://ClassLang.com) alte instrucţiuni, alte dialecte **`if-else`** (`if(C)Si; else Se; / if .. fi`), implementări optimizate etc.

```
low//Generare cod low-level

class OLD .. V{#prev V}
```

```
class NEW .. V
{
 #next V
 local L
 V:=[/@ [:V:] L/]//or simple V:=L
}


class IF C
{
 NEW EndIf, Else
 cmp C, 0//ConditionEvaluation(C)
 je Else
}

class ENDIF
{
 Else:
 EndIf:
 OLD Else, EndIf
}

class ELSE
{
 jmp EndIf
 Else:
 Else:=EndIf
}

class ELSE S
{
 ELSE
 S
}

//Exemplu test instructiune IF-ELSE
IF A
 SIA1
 IF B
  SIB
 ELSE
  SEB
 ENDIF
 SIA2
ELSE SEA1
 SEA2
ENDIF
```

5. In cadrul limbajelor protocolare, instrucțiunile sunt corelate (de exemplu pot fi de tip cerere-răspuns),
- adică au o structurare de tip frază compusă din propoziții principale şi secundare. A se implementa fraze/instrucțiuni protocolare **HTTP** (`GET`, `HEAD`, `PUT`, `POST`, `DELETE`, `TRACE`, `OPTIONS`, `CONNECT`, `PATCH`, `"200 OK"`, `"404 Not Found"` etc.) în cadrul unei aplicații server de comunicare cu un client (ce poate fi tot o aplicație sau un browser).

Metode

Metodele reprezintă moduri/stiluri de îmbinare/combinare a unor fraze pentru a descrie mesaje, raționamente, idei, algoritmi etc. În cazul limbajelor de programare metodelor le corespund combinațiile de instrucțiuni şi expresii, predefinite sau definite de utilizator în scopul procesării unor date după anumite modele/reguli. Metodele transformă într-o manieră specifică datele de intrare sau modifică unele date externe acestora prin intermediul unui set de reguli în vederea obținerii unui rezultat, îndeplinirii unui task, efectuării unor operații etc. Funcție de nivelul de procesare pot fi de tip substitutiv (utilizate la nivelul de preprocesare / procesare text ca macro-substituții) sau de tip secvențe de coduri care pot fi utilizate ca atare în anumite zone ale programelor, care pot fi împachetate în proceduri sau funcții, care pot fi atașate unor structuri/obiecte (operații) etc. Funcție de limbaj sau de contextul utilizării acestora, metodele se identifică cu: secvențele/secțiunile de cod, macro-funcțiile, procedurile, funcțiile, rutinele, subrutinele, subprogramele, programele etc.

1. Implementați şi testați într-un limbaj de programare general (C/C++ de exemplu) toate tipurile de
   - metode posibile de procesare a unor date simple (de exemplu pentru scăderea/împărțirea a două numere) - fiecare metodă combinând seturi de date de intrare/procesare diferite - plecând de la evaluarea ca atare directă a codului metodei (de procesat), utilizarea macro-funcțiilor/definițiilor, utilizarea funcțiilor/procedurilor interne (de tip inline, standard, ca argumente în alte metode), utilizarea procedurilor externe aplicației dar aflate pe acelaşi calculator (proceduri-aplicație şi librării dinamice) şi apelul codului implementat/rulat pe un alt calculator.

Gestiunea metodelor în cadrul limbajelor presupune efectuarea unor transformări speciale la intrare, ieşire şi la apelul/utilizarea acestora dacă sunt încapsulate/definite şi nu sunt plasate ca atare (ad-hoc) în zona de cod în dorită. Macro-funcțiile se implementează prin stocarea asocierii dintre argumente şi conținut, asociere ce permite la apel substituirea textuală în conținutul acestora a argumentelor declarate cu cele de apel şi plasarea textuală a conținutului rezultat. Majoritatea metodelor utilizate în practică sunt însă proceduri şi/sau funcții (interne), acestea fiind implementate prin intermediul stivei aplicație care stochează temporar (pe durata fiecărui apel) argumentele de apel, adresa de revenire şi variabilele locale. La apel, conținutul argumentelor de apel se salvează în stiva aplicației, într-o anumită ordine, funcție de tipul apelului (calling convention). După plasarea argumentelor în stivă, procesorul salvează în stivă adresa de revenire şi execută un salt către corpul metodei (funcției sau procedurii) la intrarea căreia se efectuează o serie de operații (prologue) precum salvarea unor regiştri, reținerea nivelul stivei (**sp/esp/rsp** uzual salvat în **bp/ebp/rbp**), alocarea în stivă a spațiului necesar variabilelor locale etc. La ieşire (epilogue), o parte dintre operațiile de intrare sunt inversate după care se face saltul către instrucțiunile ce urmează apelului. În cazul metodelor externe aplicației, procedeul este similar doar că apelul se efectuează prin invocarea adresei specifice locației acestora (adresa din librăria dinamică sau resursa web în cazul apelului extra sistem).

2. Implementați un program de procesare a unor metode declarate şi apelate într-un cod descris într-un
   - string sau un fişier text sau completați metodele (macro-definiții, evenimente, proceduri cu alte convenții de apel sau pentru sisteme de 16/32/64 biți etc.) din implementările ClassLang (http://ClassLang.com) următoare.

```
low//Generare cod low-level

class #define f(x) y{class f x{f.res:=y}}//Exemplu implementare #define
#define F(a,b) ((a)-(b))//Exemplu definire macro-functie F
R=K-F(Q,V-W)-Z//Exemplu utilizare/apel macro-functie F

//Exemplu definire si apel metoda procedura/functie low-level (16-biti)
S:push bp\mov bp,sp\mov ax,[bp+4]\sub ax,[bp+6]\mov sp,bp\pop bp\ret//Def
```

```
push B \ push A \ call S \ add sp, 4 \ mov C, ax//Apel C=S(A, B): C=A-B

//Exemplu definire/setare metoda eveniment hardware: Intrerupere 9 Keyboard
Int9: pusha \ in al,60h \/*..*/ mov al,20h \ out 20h,al \ popa \ iret//Def
xor ax,ax \ mov es,ax \ mov word[es:9*4],Int9 \ mov word[es:9*4+2],cs//Set

class proc P .. A{/*..*/}//Clasa declarare proceduri/functii (Prologue)
class loc .. L{/*..*/}//Clasa pentru declarare variabile locale
class return V{/*..*/}//Clasa pentru instructiunea de returnare
class endp{/*..*/}//Clasa reper de sfarsit de procedura/functie (Epilogue)

proc M A1, A2//Declararea unei metode procedura/functie
 loc V, W
 V=A1-A2
 return V
 //..
endp//Sfarsit corp metoda M

M 9, 4//Apel M tip procedura
Q=M(7, 5)//Apel M tip functie
```

3. Implementați metode extra sistem prin utilizare metodelor socket, a metodelor sistem (InternetOpen, ○ InternetOpenUrl, InternetReadFile etc.), a aplicațiilor server pentru implementare si a aplicațiilor de apel aferente (browsere, alte scripturi server etc.). Puteți completa în acest sens implementările următoare pentru IntegrativeServer descărcat de la http://WebTopEnd.com prin comunicarea SOAP (cu date XML), REST(ful) parametric sau neparametric etc.

```
//MetodaExtraSistemCod.sjs (Server JavaScript)
var A=RequestParam("A"), B=RequestParam("B"); echo(`Result=${A-B}`);

'MetodaExtraSistemCod.svb (Server Visual Basic Script)
Dim A, B: A=RequestParam("A"): B=RequestParam("B"): Echo(`Result=${A-B}`)

//MetodaExtraSistemApel.sjs (Server JavaScript)
echo(UrlRequest("http://IP-ul Vecinului/MetodaExtraSistemCod.sjs?A=5&B=2"))

'MetodaExtraSistemApel.svb (Server Visual Basic Script)
Echo(UrlRequest("http://IP-ul Vecinului/MetodaExtraSistemCod.svb?A=5&B=2"))

<!-- MetodaExtraSistemApel.html sau index.html (HTML & Client JavaScript) -->
<html><body style="font-size: 400%; color: blue;"><script>
var Result, U="http://localhost/MetodaExtraSistemCod.sjs?A=5&B=2";
function Call(){
 var S=document.createElement('script'); S.async=1; S.src=U;
 document.body.innerHTML=Result; Result=undefined;
 document.body.appendChild(S); setTimeout(Call, 1000);
}
Call();
</script></body></html>
```

4. Implementați metode combinate interne/externe, low/high level pentru comanda de la distanță a unor ○ obiecte (IoT/WoT). Exemplu: fișier script server **OnOff.sjs**: `Exec("OnOff.exe " + RequestQuery);` accesat extern ca **OnOff.sjs?x** cu `x` (`RequestQuery`) având valorile `1` (On) sau `0` (Off) și care procesează comanda prin rularea `OnOff.exe X` de tipul: http://classlang.com/?Refs=None&As=OnOff.exe&File=db+26h+0A0h+82h+0+24h+1+0BAh+0FCh+3+0EEh+0B4h+0+0CDh+16h+0C3h Identificați nivelurile de translare / propagare al acțiunilor/comenzilor/metodelor On-Off: web/sistem, extern/intern, soft/hard.

Structuri

Structurile lingvistice reprezintă descrieri detaliate/complete ale unor informaţii prin intermediul unui limbaj. Structurile informaționale reprezintă colecţii de date şi/sau metode implementate de către limbaje (tablouri, înregistrări, uniuni, variant, proprietăți, obiecte) sau de către utilizatori (vectori, matrice, tensori, mulțimi, stive, cozi, liste, dicționare, arbori) în vederea descrierii şi procesării unitare a informațiilor cu caracteristici similare în cadrul unor suprastructuri numite programe. Structurile de bază implementate în cadrul limbajelor oferă posibilitatea combinării în diferite moduri a tipurilor fundamentale (primitive) de date: boolean, caracter (ascii, unicode), întreg (reprezentat pe diferiți octeți), enumerare, real, string (secvențe de caractere), referință (către un tip oarecare, o metodă, o resursă) etc. Structurile de tip tablou (array) reprezintă colecţii de date de acelaşi tip, plasate de regulă succesiv în memorie (implementarea strictă), elementele putând fi parcurse/identificate prin intermediul unui indice (tabloul unidimensional), prin intermediul a doi indici (tablou bidimensional, elementele formând un număr de grupe, fiecare element din tabloul se identifică prin indicele grupului din care face parte şi prin indicele de poziție în grup) etc. Un tablou de grad n are nevoie de n indici pentru a referi un element, tabloul având n niveluri de grupări ierarhice a elementelor, gestiunea la nivelul limbajului a accesului la un element al tabloului presupunând în prealabil efectuarea unui calcul algebric al poziției acestuia. Structurile de tip înregistrare/articol (record, tuple, struct) reprezintă colecţii de date (numite câmpuri/componente/membri) de tipuri diferite (agregate/neomogene) sau de acelaşi tip, identificate prin nume. Structurile de tip union reprezintă colecţii de date diferite virtual suprapuse (împart aceeaşi locație), dimensiunea structurii fiind dată de elementul cu dimensiunea cea mai mare. Structura de tip variant (tagged union, discriminated union, disjoint union) reprezintă colecţii de date diferite identificabile distinct în mod dinamic, la momente diferite, gestiunea acestora în cadrul limbajelor presupunând ataşarea acestora a unor extra informaţii precum identificatorul de tip etc. Structura de tip proprietate (property) reprezintă tipul de dată cu gestiune distinctă funcție de tipul de accesare (citire/scriere), fiecare tip de accesare este intermediat de câte o metodă (get/set). Structura de tip obiect (object, class) reprezintă colecţii de date şi metode, cu diferite niveluri de accesibilitate şi moştenire. Implementarea în cadrul limbajelor presupune gestiunea datelor similar cu cazul înregistrărilor (dimensiunea unui obiect este dată de regulă de dimensiunea câmpurilor de date) la care se ataşează metodele similar unor proceduri care accesează datele obiectului, codurile de apel al unor metode speciale fiind adăugat automat la creare (constructorul), ştergere (destructorul) etc.

1. Implementați o metodă generală de procesare a elementelor unui tablou (array) de orice dimensiune,
- implementat strict, metoda primind ca argumente tabloul şi tipul de procesare (de exemplu înmulțirea cu un scalar, negativarea, afişarea elementelor indiferent de dimensiune - vector/matrice etc.). Salvați tabloul strict (blocul de memorie alocat acestuia) într-un fişier şi identificați elementele acestuia.

2. Implementați şi testați o structură de tip proprietate (property) şi/sau o structură de tip variant în orice
- limbaj (dacă limbajul are predefinite astfel de structuri se pot testa comparativ etc.).

3. Pentru a clarifica specificul implementării şi gestionării în cadrul limbajelor a structurilor de tip
- obiect, definiți o clasa de obiecte care să conțină un câmp de date (un id întreg de exemplu), constructor şi destructor care să notifice momentul apelurilor acestora (în C++ de exemplu). Declarați obiecte globale, locale (inclusiv statice) într-o procedură cu doua argumente obiect (unul apelabil prin valoare şi unul prin adresă). Apelând procedura şi analizând mesajele constructorilor şi destructorilor obiectelor a se implementa un cod echivalent cu cel generat de obiectele şi procedura anterioară utilizând o structură simplă (record) pentru câmpul de date şi doar proceduri simple echivalente cu constructorul şi destructorul obiectului, apelate corespunzător pentru a obține aceleaşi notificări (adică varianta C echivalentă translată care generează aceleaşi notificări/mesaje). Cu alte cuvinte a transla un cod obiect relevant în codul procedural echivalent, proces realizat de către orice compilator de limbaj orientat obiect deoarece limbajul maşină este unul procedural.

4. Implementați un obiect cu operatorii aferenți (indexare, concatenare, conversie etc.) gestionării unui
○ string unicode (UTF-8 şi/sau UTF-16).

5. Completați şi testați implementările ClassLang (http://ClassLang.com) următoare şi pentru alte tipuri
○ de structuri (Arrays, Records, Unions, Variants, Properties, Objects etc.)

```
//********** Low-level structures **********//

//Arrays
Ab db 1,2,3,4,5,6 \ Ab.size=$-Ab \ Ab.len=Ab.size
Ac db 'Ascii',0,0 \ Ac.size=$-Ac \ Ac.len=Ac.size
Au du 'Unicode',0 \ Au.size=$-Au \ Au.len=Au.size/2
At dt 1.5e+6,1.11 \ At.size=$-At \ At.len=At.size/10
ab rb 10, 'a' \ ab.size=$-ab \ ab.len=ab.size//==10
aw rw 11, 0xFF00 \ aw.size=$-aw \ aw.len=aw.size/2//==11
ad rd 12, -1 \ ad.size=$-ad \ ad.len=ad.size/4//==12

//Records
A:  .b db ? \  .w dw ? \  .d dd ? \ A.size=$-A//Namespace style
B: B.b db ? \ B.w dw ? \ B.d dd ? \ B.size=$-B//Explicite style
C: rb B.size \ C.size=$-C//Implicite/Indirect style (via B)

mov [A.b],1 \ mov eax,C \ mov [eax+B.b-B],3 \ mov [C+B.w-B],4//Accesare

//********** High-level structures **********//

//Data
class long .. L v{L dd v}//Definire tipul de data long
long i, j//Declarare variabile i si j de tip long

//Records
class Point V{V://Definire tipul Record/Inregistrare Point
 long V.x
 long V.y
}
Point P//Declarare variabila P de tip Point

//Objects
class Obj1 V{V://Definire tipul Obiect Obj1
 long V.l
 V.OnCreate:=Obj1.Constructor
 V.OnDestroy:=Obj1.Destructor
}
Obj1.Constructor:/*..*/ret//Implementare Constructor Obj1

class Obj2 V{//Definire tipul Obiect Obj2
 Obj1 V//Mostenire
 long V.i//Camp nou
 Obj1 V.o//Camp nou
 V.OnCreate:=Obj2.Constructor
 V.OnDestroy:=Obj2.Destructor
}
```

6. Implementați o proprietate, un obiect, un driver sau o aplicație asociată unui dispozitiv
○ hardware/periferic (tastatura, mouse, port USB/COM/LPT etc.) pentru a gestiona două proprietăți/operații/evenimente complementare de tipul get/in/read/on respectiv set/out/write/off.

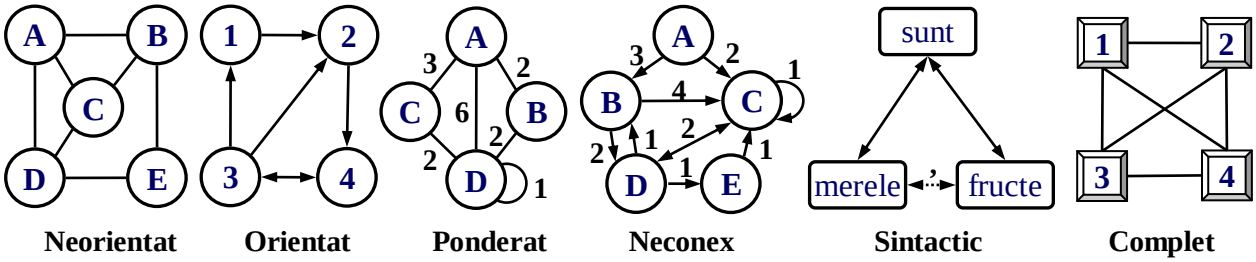
Grafuri

Grafurile reprezintă modele de descriere/reprezentare a informațiilor, structurilor, proceselor sau sistemelor compuse din elemente/entități interconectate/corelate. Grafurile sunt structuri cu valențe multiple, teoretice cât şi practice, fiind utile atât în diferite stadii de implementare sau analiză a codurilor, limbajelor şi translatoarelor, în cadrul descrierii/optimizării diferitor sisteme/procese/diagrame/circuite etc. Formal, graful se defineşte ca o pereche ordonată de mulțimi, adică prin cuplul (2-uplul):

$$G = \langle V, E \rangle$$

cu V mulțimea vârfurilor/nodurilor/punctelor (vertices/nodes/points) şi $E = \{ (i, j) \mid i, j \in V \}$ mulțimea formată din muchii/arce/legături/linii/conexiuni (edges/arches/links/lines/connections). Conectarea elementelor se poate realiza neordonat $(i, j) = (j, i) \equiv \{i, j\}$ prin muchii (grafuri neorientate) sau ordonat prin arce (grafuri orientate) dacă $(i, j) \neq (j, i)$. Un graf este ponderat dacă fiecărei muchii îi este asociată câte o valoare (pondere/weight $(i, j) = w_{ij}$). Un graf este conex dacă pentru orice două noduri, există un drum ce le leagă (conectează). Exemple (descrieri analitice/liniare/1D şi respectiv grafice/2D):

G1=⟨{A, B, C, D, E}, {{A, B}, {A, C}, {A, D}, {B, C}, {B, E}, {C, D}, {D, E}}⟩
G2=⟨{1, 2, 3, 4 }, {(1, 2), (2, 4), (3, 1), (3, 2), {3, 4}}⟩
G3=⟨{A, B, C, D}, {{A, B}=2, {A, C}=3, {A, D}=6, {B, D}=2, {C, D}=2, (D, D)=1}⟩
G4=⟨{A, B, C, D, E}, {(A,B)=3,(A,C)=2,(B,C)=4,(B,D)=2,{C}=1,{C,D}=2,(D,B)=1,(D,E)=1,(E,C)=1}⟩
G5=⟨{merele, sunt, fructe}, {{merele, sunt}, {sunt, fructe}, [{merele, fructe}=","]}⟩
G6=⟨{1, 2, 3, 4 }, {{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}}⟩=⟨{1, 2, 3, 4 }, {{1, 2, 3, 4}}⟩



Principalele operații matematice aferente grafurilor sunt: graful liniilor (graful rezultat prin schimbarea nodurilor cu muchiile), graful dual (graful rezultat prin inter-conectarea fețelor), graful invers sau complementar (generat de conexiunile lipsă), reuniunea (împachetarea şi gestionarea într-un graf a două grafuri disjuncte), produsul cartezian (înlocuirea si conectarea carteziană a fiecărui nod a unui graf cu celălalt graf), produsul tensorial (înlocuirea şi conectarea completă a fiecărui nod a unui graf cu celălalt graf). Operațiile numerice aferente grafurilor sunt: Create (creare graf), Insert (inserare nod), Remove (eliminare nod), Link (conectarea a două noduri - adăugare arc), UnLink (deconectarea a două noduri), Search (căutare nod), Explore (parcurgere/explorare graf, în adâncime – Depth First Search, în lățime – Breadth First Search), Path (căutarea unui drum dintre două noduri), Delete (ştergere graf).

Structurile de tip graf pot fi implementate în principal fie prin intermediul matricei de adiacență/pondere (adjacency matrix = matricea asociată arcelor/muchiilor, nodurile plasate într-un tablou unidimensional) sau prin intermediul listelor de adiacență (adjacency list = liste ce conțin legăturile a câte unui nod, fiecare element al listei conținând şi costul/ponderea legăturii, nodurile fiind plasate fie într-un tablou unidimensional fie alocabile individual).

1. ● Implementați şi testați operații specifice (numerice şi/sau matematice) grafurilor şi metode de verificare a căilor de cost minim, a drumurilor sau ciclurilor Hamiltonian (ce conțin toate nodurile) şi Eulerian (ce conțin toate muchiile). Grafurile test utilizate pot fi citite/salvate din/în formate TXT, CSV, TSV, XML, JSON etc.

2. ○ Reprezentați graful sintactic şi/sau lexical al unor texte literare, fraze şi/sau coduri/procese.

3. Translați circuite electronice din descrierea/limbajul NetList (TypeId Nodes Value ..) precum "R1 1 2 10k T1 2 3 4 2N22222 R2 3 5 1k V1 5 4 5V" în grafuri (scheme/diagrame 1D/2D) reprezentate liniar/analitic/tabelar/ascii/unicod/grafic (http://classlang.com/?Refs=Languages/Electronic/NetList)

4. Completați și testați implementările ClassLang (http://classlang.com) următoare ale unor grafuri utilizabile în diferite etape ale proceselor de translare, compilare şi/sau rulare: la nivel de procesare textuală (preprocesare), la nivelul codării, la nivelul execuției (rulării) etc.

```
//High-Level/Text-Processing Graph Operations
class #Insert G, iA, V{G.Node.iA:=V}
class #Link G, iA, iB, V{G.Edge.iA.iB:=V}
class #UnLink G, iA, iB{G.Edge.iA.iB:=}//#inert G.Edge.iA.iB
class #IsEdge G, iA, iB{IsEdge.res:=G.Edge.iA.iB}
class #SetNodes G, .. Node {#Insert G, selarg, Node}
class #SetEdges G, iA, .. Weight {#Link G, iA, selarg, Weight}

//Mid-Level/Code-Processing Graph Operations
class @Graph G Nodes, .. Edges{ allarg//Adjacency matrix and list/vector
 G: dd Nodes, Edges \ G.nodes=numarg
}

class @Insert G, iA, V{@set dword V at G+iA*4}
class @Link G, iA, iB, V{@set dword V at G+((1+iA)*G.nodes+iB)*4}
class @GetEdge G, iA, iB, E{@get E dword from G+((1+iA)*G.nodes+iB)*4}

//G=<{A, B, C, D}, {{A,B}=2, {A,C}=3, {A,D}=6, {B,D}=2, {C,D}=2, (D,D)=1}>
@Graph G {'A', 'B', 'C', 'D'},//Nodes: A, B, C, D
         { 0 ,  2 ,  3 ,  6 },//Edges: A-A, A-B, A-C, A-D (A adjacency list)
         { 2 ,  0 ,  0 ,  2 },//Edges: B-A, B-B, B-C, B-D (B adjacency list)
         { 3 ,  0 ,  0 ,  2 },//Edges: C-A, C-B, C-C, C-D (C adjacency list)
         { 6 ,  2 ,  2 ,  1 } //Edges: D-A, D-B, D-C, D-D (D adjacency list)

//Low-Level/Code-Running Graph Operations
class Insert G, iA, V{mov dword[G+iA*4],V}
class Link G, iA, iB, V{mov dword[G+((1+iA)*G.nodes+iB)*4], V}
class UnLink G, iA, iB{mov dword[G+((1+iA)*G.nodes+iB)*4], 0}
```

5. Afișați graful conexiunilor web după modelul următor (IntegrativeServer: http://WebTopEnd.com)

```
<!-- NetworkGraph.html sau index.html (HTML & Client JavaScript): -->
<html><body style="font-size: 200%; color: blue;"><script>
var U=["http://localhost/NetworkGraph.sjs" /*, Alte Adrese/Noduri/Statii*/];
var D=[]; for(var i in U)D[U[i]]=[];
function Send(U){var S=document.createElement('script'); S.async=1; S.src=U;
                 document.body.appendChild(S);}
function List(){var i, j, A=""; for(i in U)for(j in U)if(D[U[i]][U[j]] !=
        undefined)A+=U[i]+' &rarr; '+U[j]+'<br>'; document.body.innerHTML=A;}
function Update(){var i, j; List(); for(i in U)for(j in U){D[U[i]][U[j]]=
                 undefined; Send(U[i]+"?J="+encodeURIComponent(U[j]));}
                 setTimeout(Update,10000);}
Update();</script></body></html>

//NetworkGraph.sjs (Server JavaScript):
var J=RequestParam("J"), I="http://" + RequestHost + RequestPath;//URL/URI!
if(J.indexOf("StatieExclusa")<0)echo(`D['${I}']['${J}']=1;`);//Filtre..

'NetworkGraph.svb (Server Visual Basic Script):
Dim J, I: J=RequestParam("J"): I="http://" & RequestHost & RequestPath'URL!
If InStr(J, "StatieExclusa")=0 Then Echo(`D['${I}']['${J}']=1;`)'Filtre..
```

Gramatici

O gramatică **G** reprezintă o colecţie formată din mulţimi de simboluri şi reguli de combinare ale acestora aferente descrierii sau transmiterii unor informaţii, adică este definită de cvadruplul (4-uplul):

$$G = \langle N, \Sigma, P, S \rangle$$

în care $N=\{A, B, C, ..\}$ reprezintă alfabetul simbolurilor neterminale, $\Sigma=\{a, b, c, ..\}$ alfabetul simbolurilor terminale, $\mathbb{A}=N \cup \Sigma=\{U, V, X, Y, ..\}$ alfabetul gramaticii **G**, λ sau ε şirul vid (empty string), **S** simbolul de start, $\Sigma^*=\{u, v, x, ..\}$ mulţimea tuturor combinaţiilor simbolurilor terminale Σ, $\mathbb{A}^*=\{\alpha, \beta, \gamma, \delta, ..\}$ mulţimea tuturor cuvintelor, $\alpha \rightarrow \beta$ sau (α, β) o regulă/producţie, $P=\{\alpha \rightarrow \beta \mid \alpha, \beta \in \mathbb{A}^*\}$ mulţimea producţiilor, $\alpha \Rightarrow \beta$ regulă de derivare generală, β generat conform regulilor **P** plecând de la şirul α. Limbajul generat de gramatica **G** se notează $L(G)=\{x \mid S \Rightarrow x\}$, cu **x** şir derivat din simbolul de start **S** (formă propoziţională). Clasificarea/Ierarhia Chomsky stabileşte tipurile de gramatici relevante, după cum urmează:

- **Gramatici de tip 0 – G^0** nerestricţionate / generale (recursiv enumerabile): $\alpha \rightarrow \beta$
- **Gramatici de tip 1** – G^1 dependente de context: $\alpha A \beta \rightarrow \alpha \gamma \beta$, **A** transcris cu $\gamma \neq \lambda$ în contextul $\alpha \; .. \; \beta$ (monotone $|A| \leq |\gamma|$) sau $S \rightarrow \lambda$ dacă $S \notin \alpha\gamma\beta$.
- **Gramatici de tip 2** – G^2 independente de context (necontextuale): $A \rightarrow \gamma$, **A** transcris cu γ
- **Gramatici de tip 3** – G^3 regulate/liniare: $A \rightarrow a$ sau $A \rightarrow aB$ (dreapta) / $A \rightarrow Ba$ (stânga)

În cadrul acestei clasificări, între gramatici există relaţia $G^3 \subset G^2 \subset G^1 \subset G^0$. Fiecare tip de gramatică generează corespunzător un tip de limbaj:

- **Limbaje de tip 0 – $L^0=L(G^0)$** nerestricţionate / generale / recursiv enumerabile
- **Limbaje de tip 1** – $L^1=L(G^1)$ dependente de context descrise de maşini Turing nedeterministe
- **Limbaje de tip 2** – $L^2=L(G^2)$ independente de context nedeterministe (limbaje de programare) descrise de automate cu stivă (pushdown)
- **Limbaje de tip 3** – $L^3=L(G^3)$ regulate/liniare descrise de automate finite

Între aceste categorii de limbaje existând evident relaţia $L^3 \subset L^2 \subset L^1 \subset L^0$.

Exemplu de gramatică de tip **0**:

$G = \langle \{A, B\}, \{a, b, c\}, \{A \rightarrow aAB \mid abc, bB \rightarrow bbc, cB \rightarrow Bc\}, A \rangle$ în care
$N = \{A, B\}, \Sigma = \{a, b, c\}, P = \{A \rightarrow aAB \mid abc, bB \rightarrow bbc, cB \rightarrow Bc\}, S=A,$ şi care generează limbajul

$L = \{a^n b^n c^n \mid n \geq 1\} = \{abc, aabbcc, ..\}$

Exemplu de gramatică de tip **1**:

$G = \langle \{A, B, C, D, E\}, \{a, b, c\}, \{A \rightarrow aAB \mid abC, CB \rightarrow DB, DB \rightarrow DE, DE \rightarrow BE, BE \rightarrow BC, bB \rightarrow bbc, C \rightarrow c\}, A \rangle$ care generează de asemenea limbajul

$L = \{a^n b^n c^n \mid n \geq 1\} = \{abc, aabbcc, ..\}$

Exemplu de gramatică de tip **2**:

$G = \langle \{S\}, \{a, b\}, \{S \rightarrow aSb \mid ab\}, S \rangle$ care generează limbajul

$L = \{a^n b^n \mid n \geq 1\} = \{ab, aabb, aaabbb, ..\}$

Exemplu de gramatică de tip **3**:

G = ⟨{S, A, B}, {a, b}, {S→aA, A→aA| bB, B→bB|λ}, S⟩ care generează limbajul

L = {$a^m b^n$| m, n ≥1} = {ab, aab, abb, aabb, ..}

O gramatică de tip 2 (independentă de context) este în forma normală **Chomsky** dacă regulile sale sunt de tipul: **A→BC**, **A→a**, **S→λ**. Caracteristica este utilă în etapa de preprocesare (procesare textuală) de tip bottom-up (bottom-up/LR parsing) deoarece permite coborârea mai întâi până la nivelul fundamental/bottom (caracter / terminal) după care se procesează extinderi (up) ale structurilor mai complexe (incluzive).

O gramatică de tip 2 (independentă de context) este în forma normală **Greibach** dacă regulile sale încep cu un simbol terminal, adică sunt de tipul **A→aα**, **S→λ**. Regulile gramaticii de tip Greibach nu sunt stâng recursive. Caracteristica este utilă în etapa de procesare de tip top-down (top-down/LL parsing) asigurând oprirea după maxim n (G-stringuri) paşi.

O gramatică de tip 2 (independentă de context) este în forma normală **Operator** dacă regulile sale nu conţin consecutiv două simboluri neterminale, adică sunt de tipul **A→..BaC..**. Regulile gramaticii în forma normală **Operator** permit ancorarea expresiilor (prin intermediul simbolurilor terminale separatoare de tip **a**) corespunzând gramaticilor expresiilor (aritmetice) sau de precedenţă.

O gramatică este în forma normală **Kuroda** dacă regulile sale sunt de tipul: **AB→CD**, **A→BC|B |a/λ**. Regulile gramaticii în forma normală **Kuroda** corespund gramaticilor monotone (monotonic/ linear bounded/noncontracting grammar). Dacă **C≡A** gramatica devine de tip 1 (dependentă de context) numită şi formă normală **Penttonen**.

1. Implementaţi metode de descriere a caracteristicilor unor gramatici (mulţimile de simboluri şi reguli,
   - tipul 0..3, forma normală Chomsky/Greibach/Operator/Kuroda/Penttonen etc.) si de verificare / validare a unor expresii (forme propoziţionale) aferente limbajelor generate de acestea.

2. Descrieţi prin intermediul unor gramatici de diferite tipuri regulile de formare: a numerelor
   - întregi/reale (în diferite baze), a cuvintelor dintr-o limbă, a unor imagini, a unor structuri fractale etc.

3. Completaţi şi testaţi implementările ClassLang (http://ClassLang.com) următoare ale diferitor
   - gramatici şi ale limbajelor generate de acestea.

```
//G = <{S, A, B}, {a, b}, {S -> a A, A -> a A|b B, B -> b B|''}, S>
//L = {a^m b^n | m, n>=1} = {a b, a a b, a b b, a a b b, ..}
class E {#error "Forma propozitionala invalida!"}S, A
class S p{E}A, B
class S a? p{A p}//S -> a A
class A a? p{A p}//A -> a A
class A b? p{B p}//A -> b B
class B b? p{B p}//B -> b B
class B {#print "Forma propozitionala valida!"}//B ->''

//S b a//Invalidare (Eroare)
S a a b//Validare (Fara eroare!)
```

4. Completaţi graful conexiunilor web de la capitolul grafuri cu metode de determinare în fiecare
   - moment a gramaticii asociate regulilor de comunicare dintre staţiile din reţea.

Automate

Automatele sunt sisteme care analizează/validează cereri/inputuri/informații/limbaje. Un automat **A** se descrie prin intermediul unei colecții formată din mulțimi şi reguli de transformări succesive ale unor stări, de la o stare inițială spre stări finale, în corelație cu un input, adică este definit de 5-uplul:

$$\mathbf{A = \langle Q, \Sigma, \delta, q_0, F \rangle}$$

cu **Q** mulțimea stărilor, **Σ** alfabetul de intrare, $\mathbf{\delta: Q \times \Sigma \to Q}$ funcția de tranziție a stărilor $\mathbf{\delta(q, a) \equiv \delta_a(q)}$, $\mathbf{q_0 \in Q}$ starea inițială, $\mathbf{F \subseteq Q}$ mulțimea stărilor finale. Automatul se numeşte **determinist** dacă **δ** este funcție univocă (realizează tranziții unice, adică $\mathbf{\delta(q, a)=x \in Q}$) sau **nedeterminist** dacă **δ este** funcție multivocă/multifuncție (există tranziții multiple/alternative, adică $\mathbf{\exists \delta(q, a)=\{x, y, ..\} \subset Q}$). Un automat este **finit** dacă poate avea un număr finit de stări (**Q** este finită). Limbajul acceptat de automatul **A** se defineşte prin $\mathbf{L(A) = \{w | w \in \Sigma^*, \delta(q_0, w) \in F\}}$. Două automate sunt echivalente ($\mathbf{A_1 \Leftrightarrow A_2}$) dacă acceptă acelaşi limbaj ($\mathbf{L(A_1) \equiv L(A_2)}$). Tabelele de stare/tranziție reprezintă modalitatea tabelară de a descrie tranzițiile şi caracteristicile stărilor, pe linii - informații ale câte unei stări, pe coloane - tranzițiile sau proprietățile stării în raport cu elementele alfabetului de intrare, tipul stării (intrare, ieşire) etc. Graful de stare/tranziție reprezintă un graf orientat cu noduri şi arce etichetate cu stări respectiv tranziții. Starea inițială este marcată cu o săgeată, nodurile asociate stărilor finale sunt încercuite cu linii duble sau cu săgeți emergente.

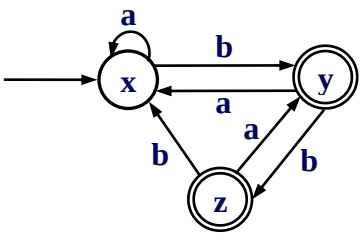
Exemplu de automat determinist

A = ⟨{x, y, z}, {a, b}, {δ(x, a)=x, δ(x, b)=y, δ(y, a)=x, δ(y, b)=z, δ(z, a)=y, δ(z, b)=x}, x, {y, z}⟩

L(A) = {b, ab, bb, aab, abb, bba, bab, aaab, aabb, abba, abab, baab, babb, bbab, bbbb, ..}

δ	a	b	In/Out
x	x	y	In
y	x	z	Out
z	y	x	Out

Tabela de stare



Graful de tranziție

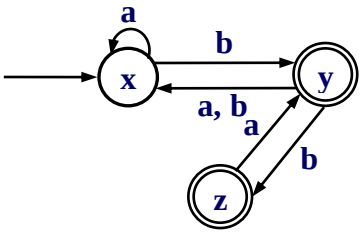
Exemplu de automat nedeterminist

A = ⟨{x, y, z}, {a, b}, {δ(x, a)=x, δ(x, b)=y, δ(y, a)=x, δ(y, b)={x, z}, δ(z, a)=y, ~~δ(z, b)=∅~~}, x, {y, z}⟩

L(A) = {b, ab, bb, aab, abb, bba, bab, bbb, aaab, aabb, abab, abbb, baab, babb, bbab, bbbb, ..}

δ	a	b	In/Out
x	x	y	In
y	x	x, z	Out
z	y	∅	Out

Tabela de stare



Graful de tranziție

Un automat pushdown **A** reprezintă un automat cu memorie gestionată ca o stivă (LIFO Last In-First Out), adică este definit prin 7-uplul:

$$A = \langle Q, \Sigma, \Gamma, \delta, q_0, z_0, F \rangle$$

cu **Q** mulțimea stărilor, **Σ** alfabetul de intrare, **Γ** alfabetul memoriei/stivei, $\delta: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to Q \times \Gamma^*$ funcția de tranziție a stărilor, $q_0 \in Q$ starea inițială, $z_0 \in \Gamma$ simbolul de start din memorie/stivă, $F \subseteq Q$ mulțimea stărilor finale. Orice 3-uplu $(q, a, z) \in Q \times \Sigma^* \times \Gamma$ se numeşte stare instantanee sau configurație a automatului.

Exemplu de automat pushdown în care

$G = \langle \{S\}, \{a, b\}, \{S \to aSb | \lambda\}, S \rangle$ este gramatica care generează limbajul

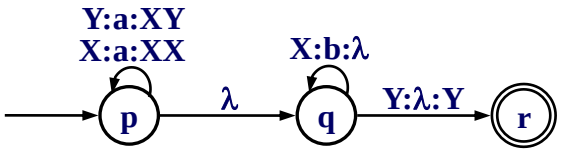
$L = \{a^n b^n | n \geq 0\} = \{\lambda, ab, aabb, ..\}$ recunoscut/acceptat de automatul

$A = \langle Q, \Sigma, \Gamma, \delta, q_0, z_0, F \rangle$ ce este format din componentele

$Q=\{p, q, r\}, \Sigma=\{a, b\}, \Gamma=\{X, Y\}, \delta=\{\delta(p, a, Y)=(p, XY), \delta(p, a, X)=(p, XX), \delta(p, \lambda, Y)=(q, Y), \delta(p, \lambda, X)=(q, X), \delta(q, b, X)=(q, \lambda), \delta(q, \lambda, Y) =(r, Y)\}, q_0=p, z_0=Y, F=\{r\}$

δ	λ	a	b
(p, Y)	(q, Y)	(p, XY)	∅
(p, X)	(q, X)	(p, XX)	∅
(q, X)	∅	∅	(q, λ)
(q, Y)	(r, Y)	∅	∅

Tabela de stare



Graful de tranziție

1. ● Implementați metode de descriere a caracteristicilor unor automate: elemente componente ale automatelor, tabele de stare, caracterul determinist sau nedeterminist, procesare/verificare/validare-invalidare a unor inputuri, listare/generare automată a unor forme propoziționale acceptate de acestea (limbajul acceptat) etc.

2. ○ Completați şi testați implementările ClassLang (http://ClassLang.com) următoare pentru diferite automate.

```
//Gramatica: G = <{S}, {a}, {S->aaS|a}, S>
//Limbaj:    L = {a^(2*n+1)|n>=0} = {a, a a a, a a a a a, ..}
//Automat:   A = <{X, Y}, {a}, {delta(X, a)=Y, delta(Y, a)=X}, X, {Y}>

class A E{#print "Reject!"}//Stare invalida, #error in loc de #print etc.
class A Y?:{#print "Accept!"}//Starea finala Y din F
class A X?: a? i{A Y: i}// delta(X, a)=Y
class A Y?: a? i{A X: i}// delta(Y, a)=X

//A X: a//Validare
//A X: b//Invalidare
//A X: a a//Invalidare
A X: a a a//Validare
//A X: a a a a//Invalidare
```

3. ○ Descrieți automatul aferent circuitului (NetList) "R1 1 2 10k T 1 3 4 2N2222 R2 3 5 1k LED 6 5 Green" (http://classlang.com/?Refs=Languages/Electronic/NetList) precum şi cel al circuitului negat.

4. ○ Implementați un automat-cifru cu cheie singulară sau multiplă de acces.

5. ○ Implementați un automat distribuit/compus pe/din mai multe calculatoare.

Maşini

O **Maşină** este un sistem compus din entităţi interdependente care transformă, în cadrul unui proces (tehnologic/informaţional/lingvistic) reversibil sau ireversibil, un input (materie primă, mişcare, forţă, energie, informaţie) într-un output. Maşinile distribuite (reţelele) conţin şi entităţi de tip maşină.

O **Maşină Abstractă** (**AM** - **A**bstract **M**achine) de calcul sau de procesare a informaţiei reprezintă un set de operaţii prin intermediul cărora un set de date abstracte/simboluri de intrare este transformat în un set de date abstracte/simboluri de ieşire. Maşina trebuie să aibă o structurare logică şi matematică self consistentă, nefiind numaidecât necesar să existe şi o structurare fizică (reală) a maşinii.

O **Maşină Turing** (**TM** - **T**uring **M**achine) reprezintă un model de procesare a unui set de simboluri în acord cu un set de reguli, fiind modelul matematic de bază al sistemelor de calcul, permiţând descrierea operaţiilor fundamentale din orice sistem de calcul. O maşină Turing este aşadar capabilă, în principiu, să descrie orice calcul algoritmic sau să realizeze orice tip de proces/transformare/translare (Conjectura Church-Turing). O Maşină Turing **M** reprezintă un model de procesare definit de 7-uplul:

$$\mathbf{M = \langle Q, \Gamma, B, \Sigma, \delta, q_0, F \rangle}$$

cu **Q** mulţimea stărilor, **Γ** alfabetul de lucru, **B∈Γ** simbolul vid (blanc), **Σ ⊆ Γ-{B}** alfabetul de intrare, **δ:Q×Γ→Q×Γ×{L, R, N}** funcţia de tranziţie/translare, cu **{L, R, N}** mulţimea deplasărilor posibile (**L**eft, **R**ight, **N**one), **q_0∈Q** starea iniţială, **F⊆Q** mulţimea stărilor finale. Cuvântul format din **αqβ**, cu **α, β ∈Γ^*** şi **q∈Q** se numeşte configuraţie / descriere instantanee. Maşina poate fi **deterministă** (**δ** univocă) sau **nedeterministă** (**δ** multivocă/multifuncţie). Fie că este deterministă sau nedeterministă sau dacă foloseşte un singur sens de deplasare sau două sau dacă foloseşte mai multe benzi (**k**) de citire/scriere simultană (**$\delta:Q\times\Gamma^k\to Q\times(\Gamma\times\{L, R, N\})^k$**), maşinile pot procesa aceleaşi limbaje, adică sunt **echipotente**.

Exemplu Maşină Turing de substituire/rescriere/ştergere simboluri **a** repetitive

Limbaj: $L=\{a^n \mid n\geq 0\}=\{\lambda, a, aa, ..\}$, Translare: $T=M(L)=\{x^n \mid n\geq 0\}=\{\lambda, x, xx, ..\}$

Maşină: M=⟨{A, B}, {a, x, s}, s, {a}, {δ(A, a)=(A, x, R), δ(A, s)=(B, s, N) }, A, {B}⟩

δ	a	x	s
A	(A, x, R)	∅	(B, s, N)
B	∅	∅	∅

Tabela de tranziţii

Graful de fluenţă

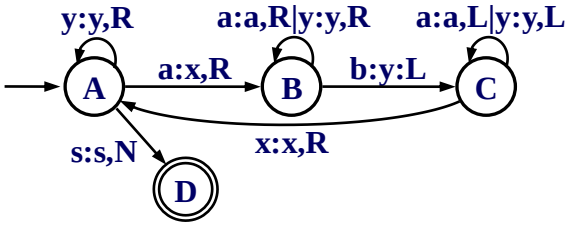
Exemplu Maşină Turing pentru procesarea limbajului **{λ, ab, aabb, ..}**

Limbaj: $L=\{a^nb^n \mid n\geq 0\}=\{\lambda, ab, aabb, ..\}$, Translare: $T=\{x^ny^n \mid n\geq 0\}=\{\lambda, xy, xxyy, ..\}$

Maşină: M=⟨{A,B,C,D},{a,b,x,y,s},s,{a,b},{δ(A,s) =(D,s,N),δ(A,y)=(A,y,R),δ(A,a)=(B,x,R),δ(B,y)= (B, y,R),δ(B,a)=(B, a, R),δ(B, b)=(C,y,L),δ(C,y)=(C, y,L),δ(C,a)=(C,a,L),δ(C,x)=(A,x,R)},A,{D}⟩

δ	a	b	x	y	s
A	(B,x,R)	∅	∅	(A,y,R)	(D,s,N)
B	(B,a,R)	(C,y,L)	∅	(B,y,R)	∅
C	(C,a,L)	∅	(A,x,R)	(C,y,L)	∅

Tabela de tranziţii



Graful de fluenţă

Un **automat liniar mărginit LBA** (**L**inear **B**ounded **A**utomaton) este o Maşină Turing nedeterministă pentru care alfabetul de intrare conţine şi două simboluri speciale **{L0, R0} ⊂ Σ** ce indică limitele de deplasare la stânga respectiv dreapta a maşinii

$$\mathbf{LBA = M = \langle Q, \Gamma, \{L0, R0\}, \Sigma, \delta, q_0, F \rangle}$$

Automatele **LBA** procesează limbaje generate de gramatici monotone**: α → β, |α| ≤ |β|**.

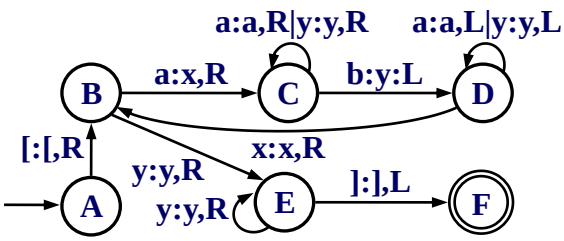
Exemplu de automat liniar mărginit

Limbaj: L={$a^n b^n$| n≥1}={ab, aabb, ..}, Translare: T={$x^n y^n$| n≥1}={xy, xxyy, ..}

Maşină: LBA=⟨{A,B,C,D,E,F},{a,b,x,y,[,]},{[,]},{a,b},{δ(A,[)=(B,[,R),δ(B,a)=(C,x,R),δ(B,y)=(E,y,R),
δ(C,a)=(C,a,R),δ(C,b)=(D,y,L),δ(C,y)=(C,y,R),δ(D,a)=(D,a,L),δ(D,x)=(B,x,R),
δ(D,y)=(D,y,L),δ(E,y)=(E,y,R),δ(E,])=(F,],R)},A,{F}⟩

δ	[	a	b	x	y	]
A	(B,[,R)	∅	∅	∅	∅	∅
B	∅	(C,x,R)	∅	∅	(E,y,R)	∅
C	∅	(C,a,R)	(D,y,L)	∅	(C,y,R)	∅
D	∅	(D,a,L)	∅	(B,x,R)	(D,y,L)	∅
E	∅	∅	∅	∅	(E,y,R)	(F,],L)

Tabela de tranziții



Graful de fluență

Maşinile de tip **Moore** stabilesc output-ul exclusiv prin starea în care se află. În cazul unor maşini de tip **Mealy** ieşirile depind atât de starea curentă cât şi de input-ul curent.

1. ● Implementați metode de descriere a caracteristicilor unor maşini (componente, tabele, caracterul determinist/nedeterminist), metode de procesare / verificare / validare-invalidare / translare / listare / generare inputuri acceptate de maşini (limbajul acceptat) şi translările efectuate de acestea.

2. ○ Construiți şi testați maşini de calcul a unor operații simple (incrementare, decrementare, dublare, înjumătățire, deplasări pe biți) pentru numere întregi reprezentate binar.

3. ○ Completați implementările ClassLang (http://ClassLang.com) următoare pentru diferite maşini

```
//Gramatica: G = <{S}, {a}, {S->aS|a}, S>, Limbaj: L={a, a a, a a a, ..}
//Masina:    M = <{A,B},{a,x,s},s,{a},{D(A,a)=(B,x,R),D(B,a)=(B,x,R)},A,{B}>
//Translare: T = M(L)={x, x x, x x x, ..}
class M C{#print [/"Failure! Configuration: ",C/]}//Stare de invalidare!
class M T/B?:s?{#print [/"Success! Translation: ",T:s/]}//Starea finala!
class M T/B?:a?:i{M T:x/B:i}// D(B, a)=(B, x, R)
class M T/A?:a?:i{M T:x/B:i}// D(A, a)=(B, x, R)
M s/A:a:a:s//Validare, Translare completa
//M s/A:a:a:c:a:s//Invalidare, Translare/Procesare incompleta/esuata
```

4. ○ Proiectați şi descrieți o maşina digitală/analogică/neuronală aferentă grafului: `G=<<R1=Rb, R2=Rc, T=BT>, <R1=T, T=R2>>` (http://ClassLang.com/?Refs=Languages/Electronic/Graph)

5. ○ Configurați o maşină distribuită după modelul următor (IntegrativeServer: http://WebTopEnd.com)

```
<!-- DistributedMachine.html sau index.html (HTML & Client JavaScript): -->
<html><body style="font-size: 400%; color: blue;"><script>
var N=0, P=0, U=["http://localhost/DistributedMachine.sjs"/*, Alte Masini*/];
function Send(U){var S=document.createElement('script'); S.async=1; S.src=U;
                 document.body.appendChild(S);}
function More(){var n; for(n in U)Send(U[n]);/*n=10000; P+=QDisk(n);N+=n;*/
          document.body.innerHTML="&pi; = "+(4*P/N); setTimeout(More, 1000);}
More();</script></body></html>
```

```
//DistributedMachine.sjs (Server JavaScript):
function QDisk(n){var p=0, x, y; while(n--){x=Math.random(); y=Math.random();
                  if(x*x+y*y<=1)p++;} return p;}
var n=10000; echo(`N+=${n}; P+=${QDisk(n)};`);
```

```
'DistributedMachine.svb (Server Visual Basic Script):
Function QDisk(n): Dim x, y: QDisk=0: Randomize: While n>0: n=n-1: x=Rnd:
         y=Rnd: If x*x+y*y<=1 Then: QDisk=QDisk+1: End If: Wend: End Function
Dim n: n=10000: Echo(`N+=${n}; P+=${QDisk(n)};`)
```

Translări

Majoritatea proceselor care au loc în cadrul sistemelor de calcul ca de altfel şi în natură reprezintă transformări ale unor informaţii structurate, reprezentate/recunoscute în cadrul unui sistem prin intermediul unor reguli sau a unui limbaj, în corespondentul informaţional specific unui alt sistem. Translarea reprezintă procesul de transformare a unor forme propoziţionale scrise într-un limbaj în formele propoziţionale corespunzătoare altui limbaj. Translările pot fi reversibile/ireversibile, ridicătoare/coborâtoare, exploratorii/euristice, de verificare/analiză, de codare/decodare, de asamblare/dezasamblare, de compilare/decompilare, de distribuire/colectare etc. In cadrul sistemelor de calcul, cele mai reprezentative translări sunt: salvarea/citirea informaţiilor (datelor/codurilor), compilarea – translarea sursei în cod maşină, decompilarea – descrierea unui cod maşină, interpretarea – translarea frază cu frază, compresia – micşorarea dimensiuni unor date, decompresia – refacerea datelor comprimate, criptarea – codarea unor propoziţii/date, decriptarea – refacerea datelor criptate, distribuirea şi colectarea datelor.

1. Implementaţi o aplicaţie care să necesite iniţializare/configurare prin citirea unor informaţii dintr-un
- fişier editabil structurat după un limbaj propriu, informaţii salvate/actualizate la închiderea aplicaţiei.

2. Implementaţi maşini/metode (interpretor/translator/compilator/codor) de procesare sau translare a
- unor coduri scrise într-un limbaj (propriu, informatic sau natural) în coduri executabile/interpretabile fie direct de către un sistem de calcul uzual fie de către o altă maşină (proprie, Java like etc.). Evaluaţi gradul de siguranţă al datelor prin intermediul informaţiei şi a entropiei acestora. De asemenea, construiţi eventual şi maşina/metoda de translare inversă (de decompilare/decodare). Se pot completa în acest sens implementările ClassLang (http://ClassLang.com) următoare.

```
//Compresie - Decompresie simboluri repetitive (self escape run-length)
class Compress n:{Compress.res:=}
class Compress n:x y z
{
 #if x==y
  Compress n+1:y z
 #elif n==0
  Compress.res:=x Compress(0:y z)
 #else
  Compress.res:=x x [*n-1*] Compress(0: y z)
 #endif
}

class Decompress {Decompress.res:=}
class Decompress x y z{Decompress.res:=x Decompress(y z)}
class Decompress x y n z
{
 #if x<>y
  failc
 #endif
 Decompress z
 #repeat n+2
  Decompress.res:=x Decompress.res
 #until
}

#print [/'',Compress(0:a a a a b b b b b # c c # # d d d # # #)/]
#print [/'',Decompress(a a 2 b b 3 # c c 0 # # 0 d d 1 # # 1)/]
```

```
//Criptare - Decriptare Caesar
class Encryption {Encryption.res:=} Decryption
class Encryption x y{Encryption.res:=[*'A'+(x-'A'+3)%26*] Encryption(y)}
class Decryption x y{Decryption.res:=[*'A'+(x-'A'-3+26)%26*] Decryption(y)}

#print Encryption('A' 'B' 'C' 'D' 'X' 'Y' 'Z')
#print Decryption('D' 'E' 'F' 'G' 'A' 'B' 'C')
```

3.○ Efectuați decompilarea şirurilor următoare de octeți (Reverse Engineering)

```
//Bios/Dos/Windows:
//bin 'exe'
db 0xb4 9 0xba 0xd 1 0xcd 0x21 0xb4 0 0xcd 0x16 0xcd 0x20 0x42 0x44 0x57 0x24

//Unix/Linux/Android:
bin ''//'elf'
db 0x7f 0x45 0x4c 0x46 1 1 1 0 0 0 0 0 0 0 0 0 2 0 3 0 1 0 0 0 0x54 0x80 4 8
db 0x34 0 0 0 0 0 0 0 0 0 0 0 0x34 0 0x20 0 1 0 0x28 0 0 0 0 0 1 0 0 0 0x54
db 0 0 0 0x54 0x80 4 8 0x54 0x80 4 8 0x23 0 0 0 0x23 0 0 0 7 0 0 0 0 0x10
db 0 0 0xb8 4 0 0 0 0xbb 1 0 0 0 0xb9 0x73 0x80 4 8 0xba 4 0 0 0 0xcd 0x80
db 0xb8 1 0 0 0 0x31 0xdb 0xcd 0x80 0x4c 0x69 0x6e 0xa
```

4. ○ Implementați după modelul următor maşini/metode de distribuire/colectare sau procesare/validare distribuită a unor date/procese/tranzacții (IntegrativeServer: http://WebTopEnd.com)

```
<!-- DistribuieColecteaza.html sau index.html (HTML & Client JavaScript): -->
<html><head><style>input, button{font-family: monospace; font-weight: bold;
font-size: 200%; color: blue;}</style></head><body>
<button onclick="Set()">Distribuie</button><input type="edit" id="SV"
value="Text de distribuit!"><br>
<button onclick="Get()">Colecteaza</button><input type="edit" id="GV">
<script>
var D={}, U=["http://localhost/DistribuieColecteaza.sjs"/*, Alte Locatii*/];
var SV=document.getElementById("SV"), GV=document.getElementById("GV"),N="N";
function Send(U){var S=document.createElement('script'); S.async=0; S.src=U;
                 document.body.appendChild(S);}
function Update(){var i; GV.value=""; for(i in D)GV.value+=D[i]; D={};}
function Set(){var i;for(i=0;i<SV.value.length;i++)Send(U[i%U.length] +
               "?M=Set&N="+N+"&V="+encodeURIComponent("D["+i+"]='" +
               SV.value.substring(i, i+1)+"';"));}
function Get(){var i; for(i in U)Send(U[i]+"?M=Get&N="+N);
               setTimeout(Update, 2000);}
</script></body></html>

//DistribuieColecteaza.sjs (Server JavaScript):
var M=RequestParam("M"), N=RequestParam("N"), V=RequestParam("V");
if(ServerBusy>70)echo(UrlRequest(`http://ReserveS/?M=${M}&N=${N}&V=${V}`));
else if(M=="Set")WriteFileContent(N, ReadFileContent(N)+V);
else if(M=="Get")echo(ReadFileContent(N));

'DistribuieColecteaza.svb (Server Visual Basic Script):
Dim M, N, V: M=RequestParam("M"): N=RequestParam("N"): V=RequestParam("V")
If ServerBusy>70 Then'Forward
                    Echo UrlRequest(`http://ReserveS/?M=${M}&N=${N}&V=${V}`)
ElseIf M="Set" Then: WriteFileContent N, ReadFileContent(N) & V
ElseIf M="Get" Then: Echo(ReadFileContent(N))
End If
```