**Curs 1**

**-> Modelul analiza-sinteza al compilarii (slide-ul 10)**

În principiu, un compilator are două mari funcții:

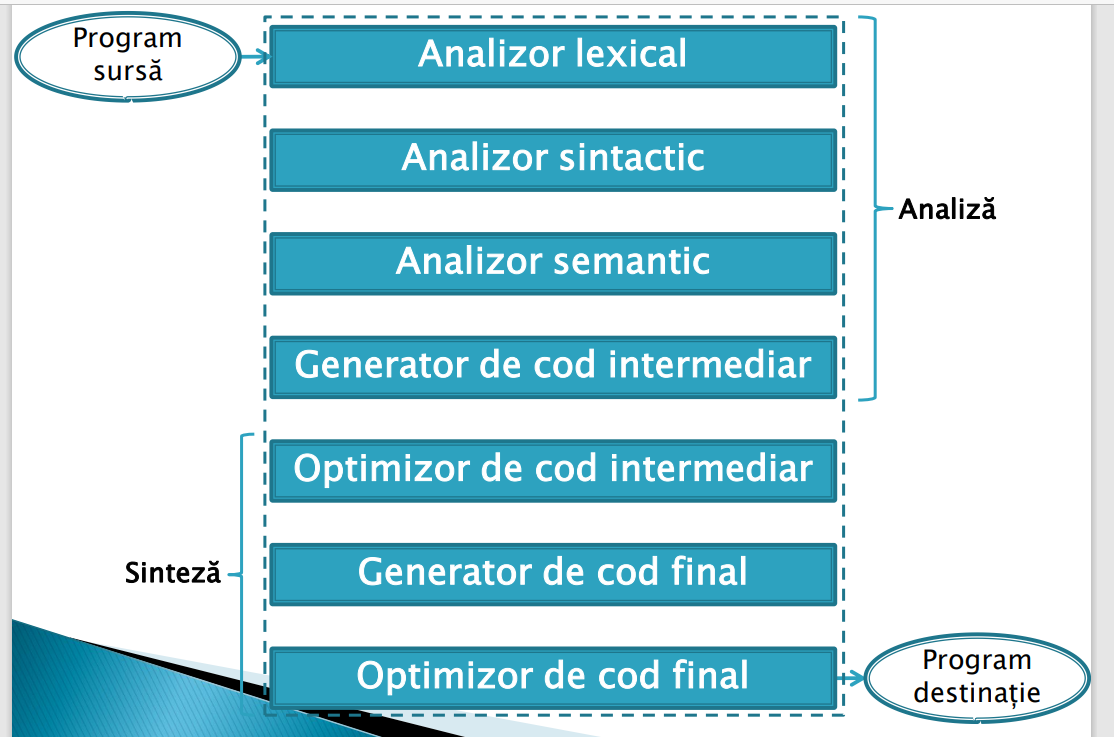
*◦ Analiza programului sursă (front-end)*

*◦ Sinteza programului destinație (back-end)*

**Analiza programului sursă** – împarte programul sursă în componentele sale de bază și, pentru compilatoarele mai avansate, creează o reprezentare intermediară a programului sursă. Uneori prelucrarea codului intermediar se consideră ca fiind o funcție separată: middle-end

**Sinteza programului destinație** – construiește programul destinație direct din informațiile de la faza de analiză sau pornind de la reprezentarea intermediară

**-> Fazele uni compilator (slide-ul 11)**



**-> din slide-ul 15 (Generatorul de cod intermediar), posibila intrebare de examen: care sunt avantajele folosirii unui cod intermediar intr-un compilator?**

Codul intermediar permite unui compilator să utilizeze același procesor de cod și back-end-uri pentru oricâte front-end-uri

**Curs 2**

**-> cum se defineste un limbaj de programare (vezi tripletul din slide-ul 2), fiecare componenta ce inseamna**

Formalizarea unui limbaj de programare (LP) se bazează pe definirea strictă atât a sintaxei cât și a semanticii sale

Un limbaj de programare este definit prin tripletul 

**St – sintaxa** - forma în care este reprezentat conținutul comunicării

**Sm – semantica** - înțelesul pe care îl are comunicarea

**F – o funcție** care asociază o semantică unei sintactici

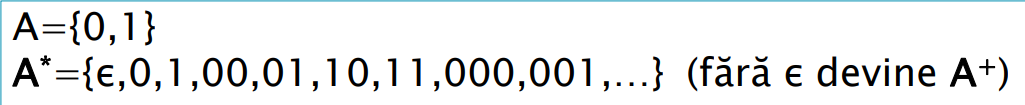
**-> alfabetul (slide-ul 3)**

**Alfabet (A)** – mulțimea finită și nevidă a simbolurilor utilizate într-un limbaj

**Propoziții** - șirurile de simboluri care se pot forma cu elementele din A

**A\*** - mulțimea tuturor propozițiilor care se pot forma cu simbolurile din alfabetul A, incluzând propoziția vidă (ε)

**A+** - A\* din care s-a exclus propoziția vidă



**-> ce este o gramatica formala si care sunt componentele ei (slide-ul 5)**

**Un limbaj formal** peste alfabetul A este o submulțime a lui A\*

Orice submulțime a lui A\* reprezintă un limbaj formal, iar A\* în ansamblu se numește limbaj universal

**Gramatica unui limbaj** – totalitatea regulilor care definesc propozițiile valide din acel limbaj

Sarcina esențială a verificării sintactice pentru un LP este de a decide dacă o anumită propoziție din A\* este conformă gramaticii acelui LP

O gramatică G= <T,N,P,s> are următoarele componente:

**T** – mulțimea tuturor terminalelor (simbolurile din alfabetul gramaticii). Vom nota terminalele cu litere mari sau între apostroafe/ghilimele.

**N** – mulțimea tuturor neterminalelor (simboluri care pot fi înlocuite prin alte șiruri, folosind regulile gramaticii). Vom nota neterminalele cu litere mici.

**P** – mulțimea tuturor regulilor (producțiilor) gramaticii

**s** – neterminalul de start.

De obicei regula de start este prima regulă din gramatică și se notează cu s, start, program,etc

Graphical user interface, text, application

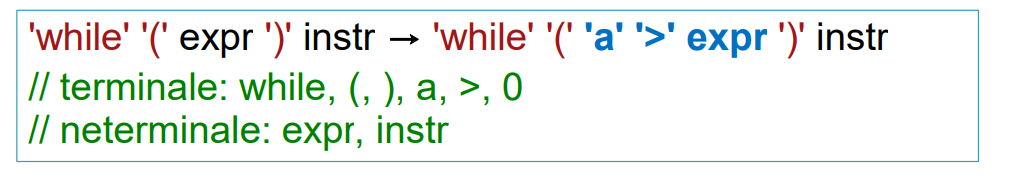
Description automatically generated

**-> ce este o regula a unei gramatici (slide-ul 6)**

O regulă este de forma **α → β** și are semnificația că prin intermediul ei șirul α va fi transformat în șirul β

**Partea stângă (α**) a unei reguli se numește cap, iar **partea dreaptă** (β) se numește coadă

De multe ori regulile se numesc producții, fiindcă prin intermediul lor se produc noi propoziții



**-> ce este o gramatica ambigua (slide-ul 19) + de dat un exemplu**

**subiect posibil** : sa ni se dea o gramatica si sa demonstram ca e ambigua

O gramatică este **ambiguă** dacă permite ca pentru aceeași propoziție să existe mai multe derivări

O gramatică ambiguă arată faptul că un limbaj nu a fost definit complet, deoarece o propoziție poate fi înțeleasă în mai multe feluri

De obicei ambiguitatea poate fi rezolvată prin transformări în gramatică

Diagram

Description automatically generated

**Curs 3**

**-> ce este comportamentul greedy al expresiilor regulate (slide-ul 12), cum se poate evita + de dat exemplu pentru acest comportament**

Implicit operatorii din ER încearcă să consume cât mai multe caractere (au comportament greedy=lacom)

Dacă sunt mai multe caractere la fel și vrem să ne oprim la primul dintre ele, atunci va trebui să scriem ER în așa fel încât caracterul de final să nu facă parte din repetiție

*Exemplu*: se dă propoziția "and", "or" și "not" sunt cuvinte în engleză și dorim să scriem o ER care să recunoască fiecare cuvânt dintre ghilimele

**Greșit: ".\*"** - va recunoaște tot subșirul de la primele până la ultimele ghilimele ("and", "or" și "not" ). Deoarece se încearcă să se consume cât de mult posibil și punctul se poate substitui inclusiv cu ghilimele, se vor consuma și ghilimele intermediare.

**Corect: "[^"]\*"** – după ghilimelele inițiale se vor consuma doar caractere care nu sunt ghilimele, până când se vor întâlni următoarele ghilimele

**Exemple**

sun|mon|tue|wed|thu|fri|sat

0?[1-9]|[12][0-9]|3[01]

[+-]?[0-9]+(\.[0-9]+)?

\/\/[^\n\r\0]\*

\'(\\.|[^\\'])\'

**Curs 4**

**-> care sunt sarcinile analizei lexicale (slide-ul 3)**

**Analiza lexicală** este prima fază a translatării (compilării)

Ea se realizează de către analizorul lexical (ANLEX)

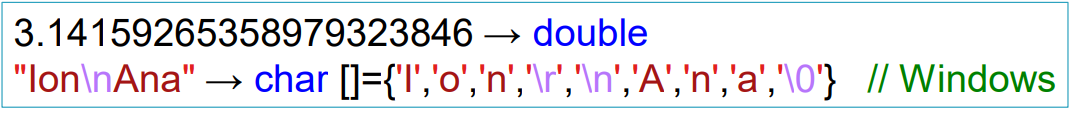
ANLEX primește la intrare textul sursă la programului sub forma unui șir de caractere și va genera un șir de atomi lexicali (tokens) corespondent. Atomii lexicali produși vor fi folosiți în faza următoare, analiza sintactică

**Sarcinile :**

-Obține șirul de atomi lexicali corespondent codului sursă

-Elimină construcțiile care nu mai sunt necesare în fazele ulterioare: comentarii, spații, linii noi,   
- Reține locațiile (linie, coloană) din codul sursă corespunzătoare atomilor lexicali pentru a se genera ulterior, dacă este nevoie, mesaje de eroare localizate

- Realizează unele transformări din caractere în alte formate, cum ar fi: constantele numerice → tipuri numerice, secvențe escape → coduri corespunzătoare, … } Separă cuvintele cheie de identificatory



**-> ce sunt definitiile regulate si care este diferenta intre definitii regulate si expresii regulate (slide-ul 9)**

Definițiile regulate (DR) se obțin din expresii regulate (ER), prin următoarele modificări:

Fiecărei ER i se va atribui un nume. Toate aceste nume formează o mulțime {d1 , d2 , …dn } Spațiile nu mai sunt semnificative

În interiorul DR pot să apară numele altor DR. Pentru a se evita recursivitatea, în interiorul DRi vor putea apărea doar numele din mulțimea {d1 , d2 , …di-1 }

Pentru a se face distincția între nume de DR și caractere obișnuite, caracterele vor fi puse în clase de caractere sau între apostroafe sau ghilimele. Din acest motiv și apostroafele și ghilimelele devin metacaractere.

**-> diagrame de tranzitii** : **fara teorie**, eventual subiect de problema in care sa facem o diagrama de tranzitii pentru cateva reguli lexicale date

Sunt o modalitate grafică de reprezentare a atomilor lexicali, folosind automate cu stări finite (ASF)

**Curs 5**

-> nimic

**Curs 6**

-> nimic

\* daca avem de implementat diagrame de tranzitii, putem alege algoritmul predat la laborator ori cel cu stari implicite

**Curs 7**

**-> rolul analizorului sintactic (slide-ul 2),**

-Pe baza gramaticii sintactice, grupează atomii lexicali în secvențe specifice limbajului: expresii, declarații, instrucțiuni, … } Alfabetul de intrare este constituit de atomii lexicali produși de ANLEX

- Secvențele recunoscute pot fi imbricate: o instrucțiune poate conține expresii sau instrucțiuni, care la rândul lor pot conține alte expresii, …

- În caz de eroare, produce mesaje de eroare localizate

-Pentru a se mări productivitatea programatorului, la apariția unei erori se poate continua compilarea, astfel încât să se depisteze cât mai multe erori la o singură compilare

**-> ce este arborele sintactic (slide-ul 4) + de dat exemplu (ne alegem reguli gramaticale si un exemplu de cod sursa pentru care trebuie construit arborele sintactic tinand cont de regulile gramaticale)**

-Este o reprezentare grafică a procesului de derivare

- Nodurile sunt regulile sintactice (neterminalele)

- Frunzele sunt atomii lexicali (terminalele)

Graphical user interface, application

Description automatically generated

**-> precedenta si asociativitate (slide-ul 6), subiect posibil: ce este precedenta si asociativitatea, enumerati cu exemple fiecare tip de asociativitate**

**Precedența** – ordinea de evaluare a operatorilor. Operatorii cu precedența mai mare se vor evalua primii.

*Exemplu:* în expresia a+b/2, împărțirea se va efectua prima (are precedența mai mare decât adunarea), deși în șirul de atomi lexicali adunarea este prima operație

**Asociativitatea** – ordinea de evaluare a operatorilor cu aceeași precedență. Asociativitatea poate fi:

- **non-asociativă** – nu se permite înlănțuirea operatorilor (exemplu: a+=b+=1 este invalid în Python)

-**stângă** – operatorii se evaluează de la stânga la dreapta

- **dreaptă** – operatorii se evaluează de la dreapta la stânga

*Exemplu*: expresia 18/6/3, în funcție de asociativitate, se va evalua astfel:

stângă: (18/6)/3 →1

dreaptă: 18/(6/3) → 9

**-> implementarea precedentei : cum se implementeaza asociativitatea dreapta (slide-ul 11) si asociativitatea stanga (slide-ul 10)**

-Precedența se implementează folosind reguli separate pentru fiecare nivel de precedență   
- **Regulile cu precedența mai mică** vor apela regulile cu precedența mai mare, astfel încât regulile cu precedență mai mare să se execute primele, și abia apoi să se revină în regulile cu precedență mai mică

**-Regulile cu precedență mai mică** vor trebui să aibă posibilitatea să se reducă la regulile cu precedență mai mare, astfel încât operatorii lor să nu fie obligatorii în expresie

Diagram

Description automatically generated with low confidence

**Asociativitatea stângă** **se implementează** folosind reguli recursive la stânga, astfel încât rădăcina arborelui va fi operatorul cel mai din dreapta și tot așa recursiv, pe subarborele stâng

Exemplu: pentru nivelul de precedență al adunării/scăderii și ținând cont că acești operatori sunt binari (arborele trebuie să aibă 2 ramuri), putem scrie

**exprAdd = exprAdd ( '+' | '-' ) exprMul | exprMul**

-exprMul este regula pentru precedența imediat mai mare.

Diagram

Description automatically generated with medium confidence

**Asociativitatea dreaptă se implementează** folosind reguli recursive la dreapta, astfel încât rădăcina arborelui va fi operatorul cel mai din stânga și tot așa recursiv, pe subarborele drept

Exemplu: pentru nivelul de precedență al atribuirii în C, putem scrie

**exprAssign = ID ‘=‘ exprAssign | exprAdd**

exprAdd este regula pentru precedența imediat mai mare.

A picture containing funnel chart

Description automatically generated

**Curs 8**

**-> la analizorul sintatic tip de problema la examen: cateva reguli sintactice si sa scriem functiile care implementeaza aceste reguli (vezi slide-ul 8) + tratarea erorilor***\*aici nu stiu exact ce ar trebui dar am pus ceva teorie legata de asta\**

Există două metode generale de implementare a unui analizor sintactic (parser):

**Analiza sintactică descendentă (top-down parsing)** – se pornește de la regula de start a gramaticii și se derivă regulile pană când se ajunge la terminale. Arborele sintactic este construit de la rădăcină către frunze.

**Analiza sintactică ascendentă (bottom-up parsing)** – se pornește de la terminale și acestea se reduc la reguli sintactice, până când se ajunge la regula de start. Arborele sintactic este construit de la frunze către rădăcină

Funcția **consume** furnizează ANSIN următorul atom lexical de prelucrat

- consume primește ca parametru un cod de atom

- Dacă atomul curent are codul cerut, el se consumă (se trece la următorul atom), iar consume va returna true

- Dacă atomul curent are alt cod decât cel cerut, consume va returna false, rămânând la poziția curentă

Graphical user interface, text, application, chat or text message

Description automatically generated

**Tratarea erorilor sintactice**

Una dintre funcțiile ANSIN este generarea de mesaje de eroare relevante, dacă a avut loc o eroare la parsarea șirului de atomi

Mesajele de eroare trebuie să fie:

**- localizate** – trebuie să conțină suficiente informații pentru a localiza de unde anume s-a generat eroarea

**-destinate celor care folosesc compilatorul** – textul mesajului de eroare trebuie conceput ca să aibă înțeles din perspectiva celor care folosesc compilatorul, nu din perspectiva celor care îl implementează

**-specifice** – secvențe diferite din gramatică trebuie să genereze erori diferite, care să permită înțelegerea regulilor care nu au fost respectate

Pentru a localiza o eroare, mesajul generat poate conține:

-numele fișierului de intrare

-linia și coloana din fișier

-marginile din fișierul de intrare ale regulii în interiorul căreia s-a generat eroarea

Pentru generarea erorilor se poate folosi o funcție (ex: *err(mesaj\_de\_eroare*) ), care să adauge automat la mesajul generat informațiile de localizare, pe baza atomului curent)

Prin analiza gramaticii se determină ce erori pot fi generate pe baza ei

Regulă generală: dacă într-un anumit punct din gramatică există posibilitatea ca analiza să se blocheze, în acel punct este posibilă o eroare.

Algoritm pentru tratarea erorilor posibile care pot apărea într-o gramatică:

- regulile în sine nu vor genera erori, ci doar vor returna false dacă nu sunt îndeplinite.

- în cadrul fiecărei reguli se determină care sunt punctele în care analiza se poate bloca

- pentru fiecare dintre aceste puncte se va prevedea un mesaj de eroare specific

**-> eliminarea recursivitatii stangi (slide-ul 12) + tip de problema la examen -> se da urmatoarea expresie sintactica (vezi slide-ul 14) si se cere: sa se scrie formula de eliminare a recursivitatii stangi (din slide-ul 13), si se se elimine recursivitatea stanga din regula data, aplicand formula.**

***\*INCEPAND CU SLIDE-UL 116 DIN PDF-UL CU 240 PAGINI\****

**Curs 9**

**-> revenirea din eroare (slide-ul 11)**

În cazul cel mai simplu de implementare, un compilator se oprește după ce raportează prima eroare

Dezavantajul acestei metode este că dacă există mai multe erori, programatorul va trebui să compileze de mai multe ori un program, pentru a le depista pe toate, ceea ce poate lua timp

Din acest motiv, un compilator poate încerca să compileze în continuare un program, pentru a raporta cât mai multe erori la o singură compilare

Pentru a compila în continuare un program, după ce a apărut o eroare, compilatorul trebuie să revină din eroarea respectivă

**-> care sunt modalitatile principale de revenire din eroare (vezi cele doua strategii din slide-ul 13)**

Există mai multe strategii de revenire din eroare, dar ele se pot împărți în două categorii:

**renunțarea la parsarea regulii curente** **și** c**onsumarea tuturor atomilor care țin de regulile dependente de ea**, până când se ajunge la un punct de sincronizare

**continuarea parsării regulii curente prin introducerea de elemente care să ducă la îndeplinirea ei cu succes**

Un compilator poate să combine mai multe strategii, în funcție de cele mai probabile cauze de apariție a erorilor:

Erorile care apar de obicei prin greșeli sintactice să fie tratate prin renunțarea la regula curentă Erorile care de obicei provin din omisiuni să fie tratate prin completarea regulii curente

**Revenirea folosind puncte de sincronizare**

La apariția unei erori se revine din toate regulile care depind de regula curentă, până la o regulă care se consideră că include toată construcția eronată, regula însăși încheindu-se în mod corect. Punctul de sincronizare este chiar atomul care încheie regula la care s-a oprit revenirea.

Se consumă toți atomii până la punctul de sincronizare inclusiv, iar apoi se reia analiza sintactică

Exemple de puncte de sincronizare: punct și virgulă, acoladă închisă, cuvinte cheie, …

**Revenirea prin completarea regulii**

Conform regulii curente, se consideră ce elemente (atomi sau reguli) trebuie să existe pentru ca regula să se completeze cu succes

Se consideră că aceste elemente există în șirul de intrare, astfel încât se continuă parsarea regulii

Exemple de elemente care se presupune a fi existente:

-după o expresie se consideră că există punct și virgulă

- după un if există paranteză deschisă

- după un break există punct și virgulă

**Curs 10**

-> nimic

**Curs 11**

**-> ce este analiza semantica si care sunt fazele ei – de domeniu si de tipuri (slide-ul 2)**

-Se ocupă cu semnificația programului

- Sintaxa stabilește forma programului, iar semantica determină conținutul său

-Pot exista mai multe forme care să aibă același conținut (ex: aproape fiecare limbaj de programare are o instrucțiune while, dar cu o sintaxă diferită)

-Analiza semantică se compune din:

**Analiza de domeniu –** analizează declarațiile din program (variabile, funcții, …)

**Analiza de tipuri** – analizează folosirea constantelor și a simbolurilor

**-> ce este arborele sintactic abstract + de dat exemplu pentru o constructie oarecare (vezi in slide-ul 3)**

**ASA** reprezintă conținutul unui program (en: AST – Abstract Syntax Tree)

ASA este o structură arborescentă de obiecte, fiecare obiect fiind o construcție de limbaj, iar copiii unui nod fiind componentele acelei construcții

Din ASA este eliminată sintaxa și astfel construcții identice reprezentate în limbaje diferite conduc la același ASA

Diagram

Description automatically generated

**Folosirea ASA**

-ASA se folosește ca o reprezentare intermediară în compilatoare, în special pentru LP mai complexe sau la care este nevoie de mai multe procesări

-ASA este util la LP care acceptă folosirea simbolurilor înainte de definirea lor (ex: Java). La aceste LP sunt necesare mai multe parcurgeri ale codului sursă, prima oară fiind pentru colectarea simbolurilor.

-Dacă se dorește generarea ASA, regulile sintactice vor cuprinde codul pentru preluarea informațiilor utile din atomii lexicali și formarea ASA cu acestea

-După ce s-a construit ASA, nu mai este nevoie de atomii lexicali, așa că se poate renunța la aceștia

-Alternativ, dacă nu se dorește folosirea ASA, în faza de analiză sintactică se pot implementa direct următoarele etape (analiză de domeniu, de tipuri, generare de cod, …)

**Implementarea ASA**

Pentru construcția ASA, se definesc structurile de date care vor conține informațiile utile din program, iar apoi se populează aceste structuri cu datele extrase din codul sursă

Structurile de date pentru ASA se pot implementa ca o ierarhie de clase, având ca bază o clasă abstractă

Fiecare tip de date din ASA (declarații, instrucțiuni, expresii) vor fi implementate prin subclase specifice

Fiecare nod din ASA va fi o instanță a unei clase

**-> ce sunt atributele sintetizate si mostenite(slide-urile 6 si 7)**

Pentru ca regulile sintactice să poată să returneze componentele utile care au fost recunoscute în cadrul analizei sintactice, este necesar să existe un mecanism prin care aceste reguli să poată returna componentele recunoscute

Uneori este nevoie ca o regulă să primească informații care deja au fost extrase de alte reguli Pentru toate aceste situații, se folosesc atribute, care sunt echivalentul parametrilor unei funcții și valorilor returnate:

**-atribute sintetizate**

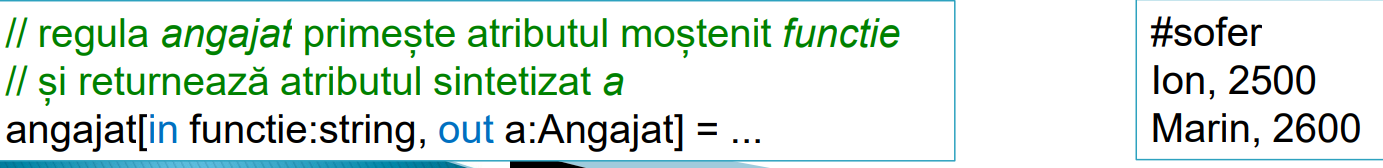
**-atribute moștenite**

**Atribute sintetizate** – valori care sunt calculate în interiorul regulii sintactice și apoi returnate de ea. Atributele sintetizate sunt echivalente valorilor returnate de o funcție.

**Atribute moștenite** – valori pe care regula sintactică le primește din exterior, în general de la regulile sintactice părinți sau frați. Atributele moștenite sunt echivalente parametrilor unei funcții.

Unei reguli sintactice i se pot asocia unul sau mai multe atribute

Atributele se scriu după numele regulii sintactice, între paranteze drepte, specificându-se pentru fiecare ce fel de atribut este (out-sintetizat, in-moștenit) și tipul său.



**Curs 12**

**-> ce este analiza de domeniu (e inclusa la analiza semantica mentionata anterior)**

Analizează structurile din program care introduc noi **simboluri**: definiții, declarații, …

**Definiție** – o structură de program care specifică în mod complet și introduce un nou simbol, posibil împreună cu conținutul său

**Declarație** – o structură de program care specifică (posibil incomplet) un simbol definit în altă parte din program

În AD se creează tabela de simboluri (TS)

TS va fi folosită atât în AD cât și în alte etape

**-> ce este contextul si domeniul unui simbol (slide-ul 4)**

Simbolurile sunt identificatorii folosiți în cadrul unui program, împreună cu semnificația și informațiile lor asociate

Simbolurile pot fi variabile, funcții, tipuri, …

Graphical user interface, text, chat or text message

Description automatically generated

**Domeniu** – toate locurile din program în care un simbol poate fi folosit

**Context** – toate simbolurile care sunt disponibile într-o anumită locație a unui program

La marea majoritate a limbajelor cu structură de bloc, un bloc definește un nou domeniu care este imbricat în domeniul părinte

**-> ce informatii pot avea asociate simbolurile (slide-ul 5)**

**-fel –** variabilă, funcție, parametru, structură, …

- **tip** – int, float[], double (\*)(double)

**-nivelul de imbricare al blocului curent** – diferențiază între variabilele globale și cele locale

**-parametrii și codul funcțiilor, câmpurile structurilor, …**

**-> cum se clasifica limbajele de programare in functie de complexitatea regulilor semantice(RS) (slide -ul 12)**

În funcție de complexitatea RS necesare pentru analiza de domeniu, putem distinge următoarele tipuri de limbaje:

**Limbaje statice cu predeclarare de simboluri** (Pascal, C) – un simbol trebuie prima oară declarat și apoi folosit (în C se permit și apeluri de funcții nedeclarate încă, presupunându-se că ulterior funcțiile vor fi definite conform unor reguli predefinite)

**Limbaje statice fără predeclarare de simboluri** (C++, C#, Java) – unele construcții (ex: în interiorul claselor) permit folosirea unui simbol care va fi declarat ulterior

**Limbaje cu declarare dinamică de simboluri** (Python, JavaScript, PHP, Ruby) – simbolurile se pot declara sau șterge chiar și în faza de execuție a programului

**Curs 13**

**->Ce este analiza de tipuri si din ce se compune ?**  
Analizează dacă simbolurile sunt folosite în concordanță cu tipul lor

La limbajele de programare cu inferență de tipuri (determinarea automată a tipului unui simbol), deduce tipurile simbolurilor

În funcție de tipul LP, AT poate fi făcută la compilare (compile time), pentru LP cu tipuri statice (statically typed) sau la execuție (runtime), pentru LP cu tipuri dinamice (dynamically typed)

Text

Description automatically generated

**-> ce sunt limbajele de programare cu tipuri statice +  caracteristici (slide-urile 3 si 4)**

-Tipurile sunt asociate simbolurilor și este cunoscut încă de la compilare

- Tipul unui simbol nu mai poate fi schimbat ulterior

- Tipurile pot fi specificate explicit, de către programator, sau se poate utiliza inferența de tipuri pentru determinarea lor automată

- Exemple: C/C++, C#, Java, OCaml, Haskell, Rust  
 **Caracteristici:  
-**Sunt mai rapide decât LP cu tipuri dinamice, deoarece AT se face încă de la compilare și toate informațiile obținute despre tipuri pot fi folosite la optimizarea codului

-Necesită mai puțină memorie la execuție, deoarece informațiile despre tipuri nu mai sunt necesare la execuție

-Se pot detecta încă de la compilare erori sau atenționări (warnings) care țin de AT

- Dacă LP nu are inferență de tipuri, în general codul sursă este mai mare, deoarece trebuie specificate explicit toate tipurile

- Dacă tipurile sunt date explicit în codul sursă, se crește inteligibilitatea codului, în special în cazurile în care este vorba despre cod scris de altcineva

**-> ce sunt limbajele de programare cu tipuri dinamice + caracteristici (slide-urile 5 si 6)**

-Tipurile sunt asociate valorilor simbolurilor, nu simbolurilor în sine

-În momente diferite un simbol poate conține valori de tipuri diferite

-Exemple: Javascript, Python, PHP, Ruby, Lisp, Prolog  
  
**Caracteristici :**

**-**Sunt mai lente decât LP cu tipuri statice, deoarece AT se face la execuție, în general înainte de fiecare operație

- Necesită mai multă memorie la execuție, deoarece fiecare valoare trebuie să aibă asociat tipul său

- Erorile specifice AT (ex: funcție apelată cu număr sau tipuri incorecte de argumente) se vor detecta doar la execuție, deci este nevoie de o testare mai complexă

-În general codul sursă este mai mic decât la LP cu tipuri statice

- Pentru a crește inteligibilitatea codului, programatorii pot specifica tipurile în comentarii sau în documentația atașată

- În general codul are tendința de a fi generic, el putând procesa valori cu tipuri diferite

**-> ce sunt valorile stanga si dreapta + de dat exemplu pentru fiecare categorie (slide-ul 7)**

**O valoare stânga** (left-value, L-value) este o locație de memorie, care poate stoca o valoare

- Valorile stânga pot apărea în partea stângă a unei operații de atribuire, dacă nu sunt constante **O valoare dreapta** (right-value, R-value) nu are asociată o locație de memorie, deci nu i se pot atribui valori

- Valorile dreapta **nu** pot apărea în partea stângă a unei operații de atribuire

Text

Description automatically generated

**-> ce sunt tipurile de date generice (slide-ul 14) + de dat  exemplu cod care foloseste date generice  (din ce limbaj dorim)**

Tipurile de date generice (template, generics) permit ca un tip în sine să fie tratat ca o constantă, a cărei valoare poate fi dată de programator

Se elimină astfel nevoia de a scrie cod foarte asemănător, atunci când codul este aproape identic la modificarea tipului de date asupra căruia acționează

Graphical user interface, text, application, chat or text message

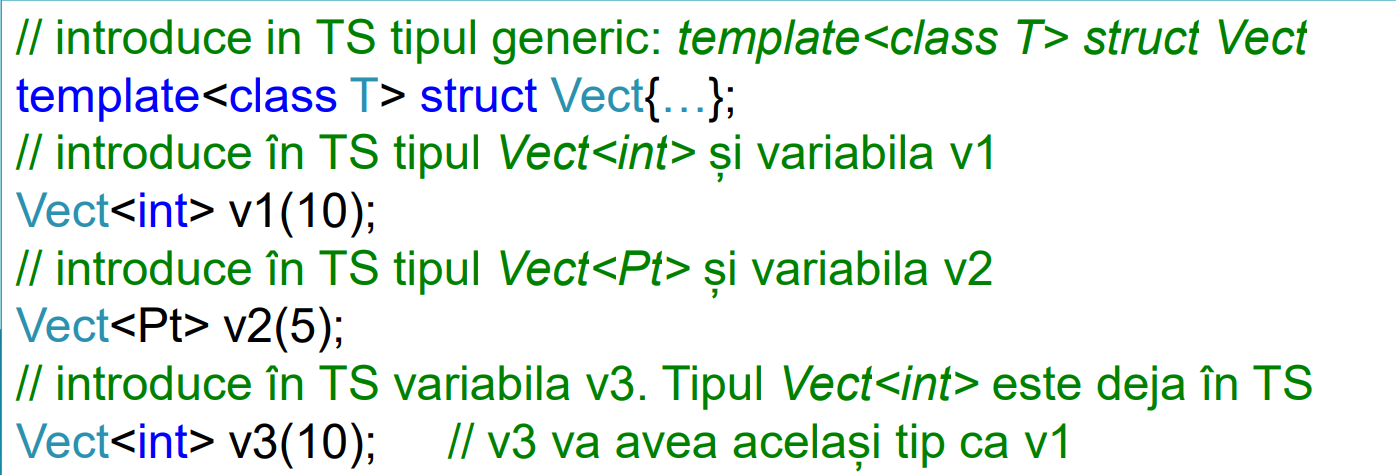
Description automatically generated

**-> analiza de tipuri pentru tipuri de date generice (slide-ul 15)**

La instanțierea unui tip de date generic (ex: Vect), se creează un nou tip de date, cu parametrii generici înlocuiți cu tipurile date la instanțiere

Simbolurile nou create (clase, variabile, funcții, …) se introduc în TS

AT va trata aceste noi simboluri la fel ca pe orice alte simboluri



**-> ce este inferenta de tipuri + categorii de tipuri sunt implementate in limbaje de programare (slide-ul 16)**

Este deducerea automată a tipului unei construcții, fără a mai fi necesar ca programatorul să scrie explicit acel tip

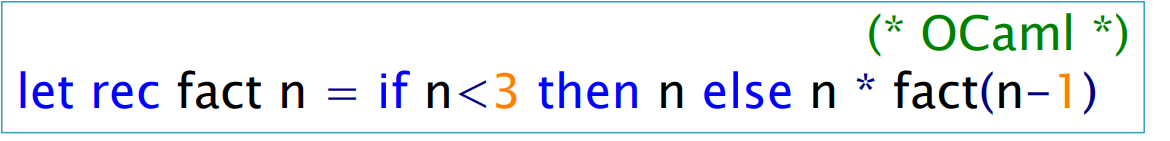
IT poate fi totală, atunci când programatorul nu mai trebuie să scrie în definiții niciun tip (ex: OCaml, Haskell) sau parțială, când LP permite inferența doar în cazul anumitor construcții (ex: C++, Rust)

Algoritmii de inferență urmăresc să propage tipurile elementelor cunoscute la elementele ale căror tip nu se cunoaște încă

Se ajunge astfel la un sistem de constrângeri, care specifică pentru fiecare tip ce condiții trebuie să îndeplinească

Prin rezolvarea acestui sistem, se determină tipurile tutor componentelor

**-> o expresie se poate sa avem (in C++ sau C#) – trebuie sa enumeram etapele posibile pentru a determina tipurile prezente in expresia data ( vezi slide-ul 17)**



-fact este o funcție care are un argument

-3 și 1 sunt constante care au tipul int

-deoarece operatorii < (mai mic) și – (scădere) trebuie să aibă operanzii de același tip, rezultă că tipul lui n este int

- deoarece la instrucțiunea if tipul expresiei then trebuie să fie identic cu tipul expresiei else, rezultă că operatorul \* (înmulțire) trebuie să fie de tip int

- deoarece operatorul \* este de tip int, ambele sale argumente trebuie să fie de tip int, deci funcția fact returnează o valoare de tip int

- Din condițiile de mai sus, tipul funcției fact este int→int